REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1990	Thesis/Dissertation

4. TITLE AND SUBTITLE
THE EFFECT OF SOFTWARE REUSABILITY ON INFORMATION THEORY BASED SOFTWARE METRICS

5. FUNDING NUMBERS

6. AUTHOR(S)
WILLIAM R. TORRES

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
AFIT Student at: Oklahoma State University

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/CI/CIA -90-61

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFIT/CI
Wright-Ptatterson AFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release IAW AFR 190-1
Distribution Unlimited
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer, Civilian Institution Programs

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS	15. NUMBER OF PAGES
	246
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED			

THE EFFECT OF SOFTWARE REUSABILITY ON INFORMATION THEORY BASED SOFTWARE METRICS

William R. Torres, Capt, USAF

Date of Degree: July, 1990 Pages in Study: 246

Degree Awarded: Master of Science
in Computing and Information Science

Institution: Oklahoma State University

ABSTRACT

The main purpose of this thesis is to theoretically investigate the effect of reusing software on metrics that are based on the entropy function of communication information theory. R. Chanon's Entropy Loading and E. Chen's Control Structure Entropy were applied to C and Ada programs obtained from the open literature. Four units of program decomposition (statement, component, module, and program) along Chanon's definition of an object were introduced to classify software reuse units. A total of three versions for each of the programs included in the study were considered (i.e., optimum reuse, intermediate reuse, and no reuse). The lines of code metric was utilized to quantify the amount of nonreusable code in each of the versions of the programs. The lines of code metric was not applied to the "reused" segments of code since they are not considered part of the effort of writing the new program. Pearson product-moment correlations were computed between the metrics studied and the lines of code metric.

Entropy loading was found to be inversely proportional to the amount of reuse present in the programs. Strong correlations were found between entropy loading and the size of the resulting new program, measured by the lines of code metric. Consequently, entropy loading can presumably provide a mechanism for selecting the optimum reuse case among different possibilities for reuse. Control structure entropy was also found to be a good indicator of reuse. Strong correlations exist between control structure entropy and the size of the resulting new program.

THE EFFECT OF SOFTWARE REUSABILITY ON INFORMATION THEORY BASED SOFTWARE METRICS

William R. Torres, Capt, USAF

Date of Degree: July, 1990 Pages in Study: 246

Degree Awarded: Master of Science
in Computing and Information Science

Institution: Oklahoma State University

ABSTRACT

The main purpose of this thesis is to theoretically investigate the effect of reusing software on metrics that are based on the entropy function of communication information theory. R. Chanon's Entropy Loading and E. Chen's Control Structure Entropy were applied to C and Ada programs obtained from the open literature. Four units of program decomposition (statement, component, module, and program) along Chanon's definition of an object were introduced to classify software reuse units. A total of three versions for each of the programs included in the study were considered (i.e., optimum reuse, intermediate reuse, and no reuse). The lines of code metric was utilized to quantify the amount of nonreusable code in each of the versions of the programs. The lines of code metric was not applied to the "reused" segments of code since they are not considered part of the effort of writing the new program. Pearson product-moment correlations were computed between the metrics studied and the lines of code metric.

Entropy loading was found to be inversely proportional to the amount of reuse present in the programs. Strong correlations were found between entropy loading and the size of the resulting new program, measured by the lines of code metric. Consequently, entropy loading can presumably provide a mechanism for selecting the optimum reuse case among different possibilities for reuse. Control structure entropy was also found to be a good indicator of reuse. Strong correlations exist between control structure entropy and the size of the resulting new program.

THE EFFECT OF SOFTWARE REUSABILITY ON INFORMATION THEORY BASED SOFTWARE METRICS

By

WILLIAM R. TORRES

Bachelor of Science

in Electrical Engineering

University of Puerto Rico

Mayaguez, Puerto Rico

1985

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
July, 1990

THE EFFECT OF SOFTWARE REUSABILITY

ON INFORMATION THEORY BASED

SOFTWARE METRICS

Thesis Approved:

_____

Thesis Adviser

_____

_____

_____

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express sincere appreciation to Dr. Mansur H. Samadzadeh for his encouragement and advice throughout my thesis research. His dedication and attention to detail made all the work required to complete this thesis worthwhile. Many thanks also go to Dr. G. E. Hedrick and Dr. John P. Chandler for serving on my graduate committee.

My wife, Maria, and my daughters Maria C. and Diana B., encouraged and supported me all the way and kept me motivated to achieve the end goal. Their love and support made everything much easier. I also thank my parents, William and Nelida, and my In Laws, Jose and Marian Setien, for their support, encouragement, and many prayers.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The notion of reusing software dates back to the early stages of the history of computing when subroutine libraries were developed [SOMME89].

The advantages and limitations of software reuse have been widely publicized [BIGGE87, BOLDY89, PRIET87, RATCL87, and SOMME89]. In particular, it is clear that the extensive reuse of software is likely to reduce software costs during the design and implementation phase (software already exists) and during the validation phases (software has already been checked) [SOMME89].

The main purpose of this thesis is to theoretically investigate the application of software quality assessment metrics to development environments that employ reuse techniques and principles in the construction of software systems. In particular, the effect of reusing software on the metrics that are based on the entropy function of communication information theory will be investigated.

The following chapters define the metrics used in this study, describe the experimental design including how the datawere collected, discuss the analysis of the measure-

ments, and summarize and conclude with recommendations for future work.

CHAPTER II

LITERATURE REVIEW

2.1 Entropy Function of Information Theory

2.1.1 Entropy and the Software Development Process

In communication information theory, information is defined as "what we don't know about what is going to happen next" [SHANN64]. Information theory presents entropy as a synonym for information uncertainty or unpredictability. Therefore, entropy can be used as a measure of information.

When an attempt is made to predict the outcome of an event, uncertainty approaches zero as the probability of an outcome approaches unity. On the other hand, uncertainty, as a function of the number of different things that can happen, reaches its maximum value when all the outcomes have the same probability of occurrence (i.e., in the case of equiprobable events). In other words, the more unpredictable the output of a system, the more information it yields upon occurrence and the higher the entropy of the system, and vice versa. The entropy function, H, which measures information in bits is defined as

$$H(p_1, p_2, \ldots, p_n) = \sum_{j=1}^{n} p_j \log_2 (1/p_j) \qquad (1)$$

where pj is the probability of occurrence of the jth event.

van Emden [VANEM70] defines a system as a set of variables influencing one another. A challenge often faced by software designers when designing a system is managing the complexity resulting from the presence of a large number of variables and the fact that most of them influence many others. To alleviate this problem, Alexander [ALEXA64] states that a system composed of a set of variables, S, should be partitioned into subsets or subsystems $(S_1, S_2, \ldots, S_j, \ldots, S_k)$ in such a way that the interactions[1] among subsystems should be minimized with respect to the interactions within a subsystem. This partitioning process allows for the variables in S_j to be manipulated freely without constraints imposed by any of the other subsets.

If a program structure could be decomposed into distinct classes of subsystems, the information contained in the structure can be measured [MOHAN79, MOHAN81, and VANEM70] by the entropy metric, H, based on Shannon's communication information theory [SHANN64]. More specifically, the entropy metric H is defined for a system S as

$$H(S) = H(P_1, \ldots, P_n) = \sum_{j=1}^{n} ((|X_j|/|X|)\log_2(|X|/|X_j|))$$

---

[1]An interaction is viewed as an information transfer within or among subsystems [ALEXA64].

or by simple algebraic manipulation

$$H(S) = \log_2|X| - 1/|X| \sum_{j=1}^{n} (|X_j| \log_2 |X_j|) \qquad (2)$$

where X_1, X_2, ..., X_n are the distinct classes of subsystems, $|X_j|$ denotes the cardinality of the set X_j, and $P_j = |X_j|/|X|$.

2.1.2 Information Theory Based Metrics

Software metrics are usually categorized as either process metrics or products metrics. Process metrics are defined by Conte et al. [CONTE86] as "metrics that quantify attributes of the development process and of the development environment." Conte et al. also define product metrics as "measures of the software product."

Product metrics are the most common metrics since they are easier to obtain as they "can be derived from analyzing the software itself using an automatic tool" [CONTE86]. Examples of product metrics include size of the product (lines of code), logic structure complexity (e.g., flow control and depth of nesting), and data structure complexity (number of variables used) [CONTE86].

Information theory based metrics can be classified as product metrics. The following subsections discuss various information theory based metrics.

2.1.2.1 Entropy Loading. Section 2.1.1 described how we can use the entropy metric to measure the information

contained in a system. Using the same basic principles, we can use the entropy loading measure, as described by Chanon [CHANO73], "to measure the information shared among subsystems, as opposed to the information contained in each subsystem."

According to Chanon [CHANO73], the term interaction, means "a shared assumption among two or more objects" where objects are defined as "an identified portion of a program that has net effects." The following is a list of assumptions identified by Chanon.

1. Relationships that must hold prior to the execution of an object in order for its effects to be realized.

2. Data structures or data values.

3. Assumptions about the environment in which an object is executed, such as frequency of usage or order of computation.

4. Assumptions based on mathematical theorems that are relevant to the problems being solved.

Once all the assumptions have been identified, they can be recorded in what Chanon calls an object/assumption table. For a given program, such a table, T, is defined for all objects, I, and assumptions, J, such that

$$T(I,J) = \begin{cases} 1, & \text{if object I makes assumption J} \\ 0, & \text{otherwise} \end{cases}$$

According to van Emden [VANEM70] and as demonstrated by Chanon [CHANO73], the data contained in the object/assumption table characterizes the extent to which collections of

objects interact. This data is used in the calculation of a measure which van Emden and Chanon call entropy loading. Entropy loading is defined for a set of rows, S, in an object/assumption table at a given time in the development of a system.

Assume that S is partitioned into subsets A and B such that $A \cap B = 0$ and $A \cup B = S$. Then C(S), the entropy loading of S, is given by

$$C(S) = H(A) + H(B) - H(S) \tag{3}$$

where H(X) is given by the entropy metric mentioned in Equation (2).

Suppose the table T of objects and assumptions is given as

		a	b	c	d	e	f
S	A: 1	1	0	0	0	0	0
	A: 2	0	0	0	0	0	1
	A: 3	0	0	0	1	0	0
	B: 4	0	1	0	0	1	1
	B: 5	0	0	1	0	1	1
	B: 6	0	0	0	0	1	0

where A and B are subsets of the object set S, and a through f are the assumptions. For example, for row 1, the entry 1 indicates that object 1 (an element of A) makes one assumption, a. H(A) is computed by considering the columns for which the entry for any object of A is 1, i.e., a, d, and f. In the submatrix composed of the columns a, d, and f, (100) occurs once, (001) occurs three times, (010) occurs once, and (000) occurs once. The total number of objects is six. Therefore, using Equation (2), we have

$$H(A) = \log_2 6 - 1/6[1\log_2 1 + 3\log_2 3 + 1\log_2 1 + 1\log_2 1]$$

or

$$H(A) = \log_2 6 - 1/6[3\log_2 3] = 1.79$$

Similarly,

$$H(B) = \log_2 6 - 1/6[2\log_2 2] = 2.25$$

Since 1 occurs in each column at least once and each row is different, thus

$$H(S) = \log_2 6 = 2.58$$

and

$$C(S) = H(A) + H(B) - H(S) = 1.46$$

Chanon [CHANO73] demonstrated that programs that possess small entropy loadings also possess properties consistent with the principles of "good" structure stated by Alexander [ALEXA64], Dijkstra [DIJKS72], Parnas [PARNA71], and Simon [SIMON62] as follows:

> 1. The information required to study, understand, and verify individual parts of a system is supplied in conjunction with those parts, and relatively little information about the rest of the system is required.
>
> 2. Single parts can be changed drastically (algorithms and/or data structures) without requiring much knowledge of the rest of the system and without changing the rest of the system; i.e., drastic changes can actually be confined to single parts.
>
> 3. Should an error occur as a result of the failure of one small part to function correctly, the error can be localized to that part of the system quickly and easily, permitting the error to be repaired using only a knowledge of that part.
>
> 4. During system construction, each working group can be given an assignment to write a set of parts such that the assignment can be completed with very little communication between the groups.

Based on the aforementioned discussion, entropy loading can be used as a quality assessment metric to determine how "good" a program is, when compared to other input/output equivalent programs or different versions of the same program.

2.1.2.2 Control Structure Entropy. Chen [CHEN78] defines a measure of program control complexity based on an information theoretic viewpoint. Given a strongly-connected proper flow chart[2] which results from structured programming, Chen defines the maximal intersect number, MIN, as "the maximum number of edges which can be intersected by a continuous line drawn such that the line never enters any region, including the external region, more than once." Chen utilizes three basic types of control structures [CHEN78] to obtain MIN from the charts: 1) SEQUENCE, 2) IF p THEN f ELSE g, and 3) WHILE p DO f. These structures are shown in Figure 1.a.

If a flow chart or graph is not strongly connected, but can be visualized as consisting of more than one strongly-connected subparts connected in series, MIN can be obtained from

$$MIN = \sum_{i=1}^{n} MIN_i - 2n + 2 \qquad (4)$$

[2]A flow chart is strongly connected if there is a path from node a to node b for every pair of distinct nodes a and b [CONTE86]. A flow chart is proper if each node can be reached from the entry point of the program and if each control structure has only one entry and one exit [SHOOM83].

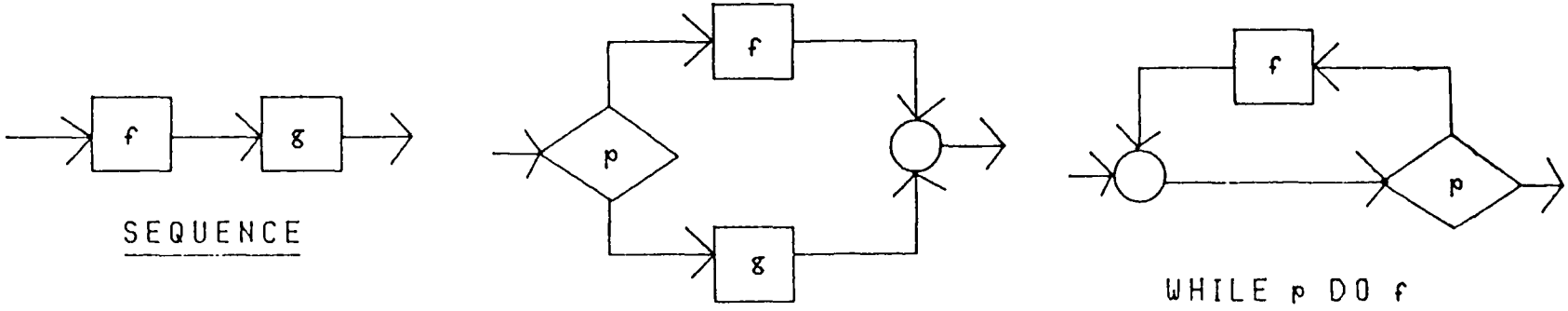
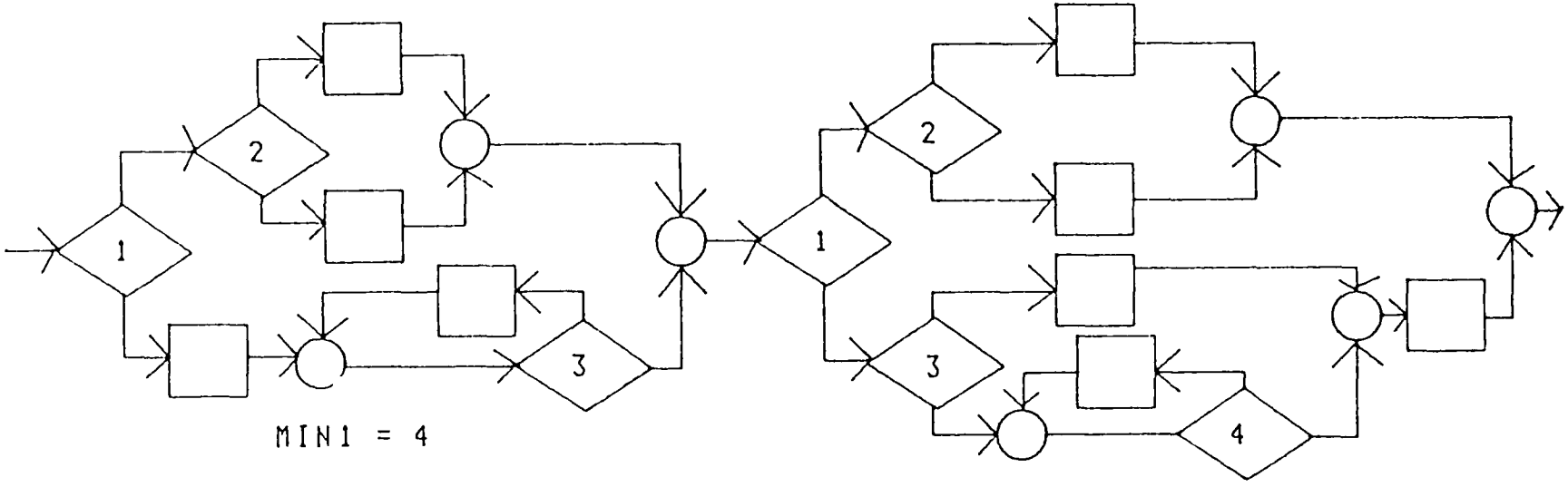


Figure 1.a



Total MIN = 4 + 5 - 2(2) + 2 = 7

Figure 1.b

Figure 1. Maximal Intersect Number Examples.

where n is the number of strongly-connected subparts and MINi is the MIN for the ith strongly-connected subpart. Figure 1.b illustrates an example obtained from [CHEN78] in which two weakly connected subparts with MIN's equal to 4 and 5, respectively, yield a MIN equal to 7 for the combined chart (i.e., 4 + 5 - 2(2) + 2 = 7).

The attribute MIN can also be computed analytically from the expression MIN = Z_n + 1 where Z_n is given by the following formula

$$Z_n = 1 + \sum_{j=2}^{n} \log_2(2p_j + q_j) \qquad (5)$$

where n is the number of decision symbols on the flow chart or graph, q_j is the probability that the j^{th} IF symbol is forming a serial relation with any of its preceding and adjointed[3] IF symbols, and $p_j = 1 - q_j$. For a given flow chart, q_j is either 1 or 0 depending on whether it is in the specified serial relationship. Considering the left subpart in Figure 1.b (three IF symbols), we have that $p_2 = p_3 = 1$ and $q_2 = q_3 = 0$ since neither IF symbol is serially connected to their preceding IF symbol. Consequently we have that

$$Z_3 = 1 + \log_2(2 + 0) + \log_2(2 + 0) = 1 + 1 + 1 = 3$$

thus

$$MIN = Z_3 + 1 = 3 + 1 = 4$$

---

[3]Adjointed IF symbols are two IF symbols which can be connected without passing through any vertices (nodes) which belong to a third IF symbol [CHEN78].

which is the same result obtained earlier.

Chen describes a programmer to be (in information-theoretic terms [SHANN64]) a "channel" who is to handle information from a program specification or problem statement described as the "source." A source of information is characterized by the variety of output symbols that it produces. In this case, the output is a sequence of IF symbols. The other components of a proper flow chart, i.e., function nodes and collecting nodes are considered and ignored because function nodes do not contribute to the branching or nesting structure of a flow graph and collecting nodes are merely converging points for the branches of the corresponding decision points.

Z_n is defined as the control structure entropy of an information source when it emits n IF symbols. This control structure entropy is a measure of the control variety of a source's output and is correlated with the conceptual complexity of the program [CHEN78]. The higher the Z_n's value, the more complex the program is. In fact, MIN was originally proposed to address some of the deficiencies of McCabe's cyclomatic complexity measure [MCCAB76].

For a given total number of IF symbols, the programming job can be modeled [CHEN78] by the task of determining the exact flow chart structure which will determine the p_j's and q_j's in Equation (5). An approximation to Equation (5) can be obtained by assuming $p_j = q_j = 1/2$ [CHEN78]. Substituting these values in Equation (5), we have

$Z_n \cong (n - 1)\log_2 3 - n + 2$. Given the number of IF symbols that a program is to have, Z_n can be calculated easily since it only depends upon n.

Based on the work by Chen [CHEN78], control structure entropy can be used as a quality assessrent metric to determine how "complex" a program is, when compared to other programs or different versions of the same program.

2.1.2.3 Additional Metrics. In addition to the entropy loading and control structure entropy metrics, other information theory based software metrics have been proposed by, among others, E. Berlinger, M. H. Samadzadeh and W. R. Edwards, and T. T. Lee.

Berlinger [BERLI80] proposed the following information theory based complexity measure. Given a program, he counts the frequency, f, of all tokens in the program. Berlinger also assumed that the probabilities of the occurrence of the tokens used in the program are known. The complexity measure, M, is defined as

$$M = -\sum_{j=1}^{n} f_j \log_2 p_j \qquad (6)$$

where n is the total number of distinct tokens, and f_j and p_j are the frequency and probability of occurrence of the j^{th} token, respectively.

Berlinger [BERLI80] presents two interpretations for this measure. First, from an information theory point of view, assuming the program to be a message, the measure

represents the total information contained in the program. Second, the measure represents the total length in bits required to encode the program assuming that each token is to be coded using a uniform encoding scheme.

One problem with this measure is that the p_j's need to be accumulated over a period of time at the installation where the new program is to be written. Consequently, the measure is of little help to a programmer at a different installation where the possibility of a completely different set of pj's exists. This makes two purportedly similar programs, written in two different installation, incomparable.

Samadzadeh and Edwards presented a model that regards the understanding process of software as a process of grouping together the tokens of a software document (either distinct tokens or all of the tokens) that have certain characteristics in common [SAMAD88]. According to the authors, "this model captures the amount and some of the structure of the information present in the software document." A simple example is the classification of program tokens into operators and operands, as originally used by Halstead [HALST79].

The measure, denoted as R, is called residual complexity. This measure is based on the difference between the maximum value of the computational work and the computational work, w, of a partition, π which is the result of classifying a piece of software into q token types. The

residual complexity can be obtained from the equation

$$R = N \log_2 N - N H(N_1/N, N_2/N, \ldots, N_q/N) \quad (7)$$

where N is the total number of tokens, N_i is the number of tokens in the i^{th} block of the partition, and H is the entropy function.

Lastly, Lee [LEE87] uses Shannon's information theoretic entropy metric to quantify the information associated with a set of attributes. An attribute is a symbol taken from a finite set $\Omega = (A_1, \ldots, A_n)$ [LEE87]. For each attribute A_j, $1 \leq j \leq n$, there is a set of possible values, denoted $DOM(A_j)$, which comprise its domain. Database design is based on the concept of data dependency, which is the interrelationship between data contained in various sets of attributes [LEE87]. In particular, Lee [LEE87] states that functional, multivalued and acyclic join dependencies play an essential role in the design of database schemas. Lee proves that data dependencies can be formulated in terms of entropies making the numerical computation and testing of data dependencies feasible.

2.2 Software Reusability

The notion of software reuse has been around since the early stages of the history of computing when the main motivation for the development of subroutine libraries was software reuse [SOMME89]. Software reuse has also been associated with software portability. According to Sommerville [SOMME89], "porting a program to another

computer can be considered an example of software reuse although it is possible to reuse a program which is not portable and can only run on a single computer."

2.2.1 Definitions

Reusability is, as defined by Wegner [WEGNE83], "a general engineering principle whose importance derives from the desire to avoid duplication and capture commonality in undertaking classes of inherently similar tasks." In the title and body of this thesis the notion of reusability has been used with Wegner's definition in mind.

Prieto-Díaz [PRIET87] defines reuse as "the use of previously acquired concepts and objects in a new situation" and reusability as "a measure of the ease with which one can use those previous concepts and objects in the new situation." Consequently, in order for software reusability to be beneficial, the effort to reuse a piece of software needs to be smaller than the effort required to develop the software from scratch.

Conceptual complexity and size are two common and related software problems. As these problems continue to loom larger, software reuse and reusability is starting to be looked upon as a possible solution; as demonstrated by the increasing attention it has received in the last seven to ten years. In spite of this fact, not a lot of effort has been devoted to standardizing and implementing reuse by major companies in the USA. On the other hand, Matsumoto

[MATSU84] demonstrated that Japanese industries had been reusing software for the past six to seven years.

2.2.2 Advantages and Limitations

When we think of software reuse, one of the first things that comes to mind is the reductions in cost as the number of components that must be specified, designed, implemented, and tested is reduced. However, Sommerville [SOMME89] states that "it is difficult to quantify the actual cost reductions attained by reusing software; if in fact there are any." The number of applications where code can be reused without any modification whatsoever is very small. Nevertheless, cost reduction is not the only advantage of software reusability. Some of the other advantages of software reusability are mentioned below.

Software reuse increases system reliability [LUBAR86b and SOMMER89]. It is widely accepted that operational use adequately tests software components, and reused components, which have been previously in operational use, should be more reliable than brand new components. Also, if a component is originally developed to be reusable, Freeman et al. and Lubars [FREEM83 and LUBAR86b] state that "the debugging costs can be amortized among the products that reuse the component."

As a result of software reuse, programming resource utilization can be improved [SOMMER89]. The availability of reusable software allows for a better distribution of

programming resources since not all the code needed is to be developed anew. Sommerville states that "application specialists can develop higher reusable components that encapsulate their knowledge."

Another advantage is the reduction in software development time [SOMMER89]. Reusing components speeds up system production because both development and validation time should be reduced.

However, as expected, software reusability is not a perfect science and there are limitations that must be kept in mind. This is specially true since some researchers think that the limitations and disadvantages outweigh the advantages.

The first problem is what to do with a piece of software once it has been determined that it is a candidate for reuse. In a recent article, Tracz [TRACZ88] presents his "Golden Rules of Reusability." Tracz says that "before you can reuse something, you need to find it, know what it does, and know how to reuse it", which go along with the need for means of cataloging, classifying, and retrieving software components.

Lubars [LUBAR86a] describes the problem of finding a desired piece of reusable code as the most significant technical barrier to code reusability. The problem of software classification has been addressed by Prieto-Diaz and Freeman [PRIET87] where they attempt to rank reusable software components using a reuse effort estimation metric.

Another problem is the not-developed-here syndrome experienced by some programmers and reflected in some company policies that do not allow non-local programs to be utilized. They prefer to write their own code because they believe that they can improve on the reusable component. But even if this is true, Cheatham and Sommerville [CHEAT83 and SOMMER89] state that "it is at the expense of greater risks and higher costs."

The last limitation we are concerned with involves the reusable code itself. Researchers are concerned about how specific or, on the other hand, abstract the code needs to be before reuse pays off. This topic is discussed in the next section on current trends.

2.2.3 Current Trends

There are two main schools of thought on software reusability [BIGGE87 and PRIET87]. The first one promotes the reuse of ideas and knowledge acquired while developing software, while the other promotes the reuse of particular artifacts and components. Although the second approach is more popular, researchers disagree on how abstract the code needs to be before it may be reused. Kernighan [KERNI84] presents reuse at the program level utilizing the UNIX[4] pipe.

---

[4]UNIX is a Trademark of AT&T Bell Laboratories.

On the other hand, Matsumoto [MATSU84] promotes reusing modules defined in higher levels of abstraction to increase the scope of the reusable code. According to Matsumoto, "there are four levels of specification: requirements, design, program, and source code." A module, when originally conceived, is transformed from the requirements level, into the design level, the program level, and finally into the source code level. When a module is to be reused, the requirements level of the new module is compared to the requirements levels of previously designed modules and when a match is found, a trace is made from the requirements level to the source code level through previously made transformations to reuse the source code.

Other researchers that promote the use of higher abstract levels for software reusability include Kaiser and Garlan, Goguen, and Cheng et al. Kaiser and Garlan [KAISE87] promote the use of Meld, "a language that blends a package library, automatic software generation, and object-oriented programming approaches to reusability." In their approach, a software system that needs to be developed is written in Meld and then translated into the desired implementation language.

Goguen [GOGUE86] presents a "library interconnection language," called LIL, to assemble large programs from existing entities by combining Ada[5] programming language

---

[5]Ada is a Trademark of the U.S. Department of Defense (Ada Joint Program Office).

specification parts with commands for interconnecting components to form systems. In a somewhat different approach, Cheng, Lock, and Prywes [CHENG84] present a very high level language, Model, which acts as a program generator to allow the nonprogrammer professional to design a system by solely describing the data interrelationships without referencing any computer operations. The result, according to Cheng et al., is "a language that is free of the conventional programming control and flow concepts, and is thus simpler and easier to use."

CHAPTER III

DESCRIPTION OF THE EXPERIMENT

3.1 Design Approach

3.1.1 Introduction

In this study the intent is to explore theoretically the effects of software reusability on Chanon's entropy loading metric [CHANO73] and Chen's control structure entropy [CHEN78]. But before either calculation can be applied, we need to establish some guidelines for identification of the "program parts" that are to be reused. This is needed because a program part is not defined in Chanon's definition of an object [CHANO73], in fact, it is left, apparently intentionally, an unspecific and generic concept (See Chapter II for a discussion of Chanon's entropy loading metric).

3.1.2 Reuse Candidates

Even though program decomposition or partitioning is, in general, language and application dependent, we define four units of program decomposition which are objects that can be considered to be along Chanon's object definition,

are visible[1] to one another, and can be candidates for reuse. These units of decomposition are statements, components, modules, and the obvious one-block partition, the program itself. The definitions of the four units of decomposition follow.

The first unit of decomposition is a statement which is the lowest level at which reuse will be considered and is defined as any executable instruction of a program which makes assumptions about, and is visible to, any other object in the program. Executable instructions that do not make assumptions are not considered statements, e.g., NEW_LINE in Ada or printf("\n") in C. These instructions do not depend on any other instruction for their execution.

The unit of the next higher level decomposition is a component. A component is a collection of one or more contiguous statements having a name and represented by the implementation of a procedure or algorithm. None of the statements in a component are visible to any other object. The only assumptions that a component can make are about the parameters passed to it when invoked or the visibility of other objects and/or global variables. Examples of components are functions in C, and procedures and functions in Ada.

A module is a unit of decomposition or a candidate for

---

[1]An object, a, is visible to any other object, b, if b is in control of program execution and control of execution can be transferred from object b to a.

reuse above the component level. A module is a collection of components which also has a name and can be invoked by any other object. Analogously to the component, none of the components inside a module are visible to any other object. The only assumptions that a module can make are about the parameters passed to it when invoked or the visibility of other objects and/or global variables. Examples of modules are procedures and packages in Ada and functions in C.

The highest level of decomposition or reuse is obviously the program itself. An entire program is the highest level at which reuse can occur. A program is defined as a collection of modules. It is an extreme case since it is the only block in the partition.

The four definitions offered above for an object are not supposed to be rigid prescriptive units of reuse. They are merely the easily recognizable milestones along the decomposition spectrum. In other words, a candidate for reuse can consist of a mixture of the above-mentioned units.

Other possibilities for reuse candidates are plans and delocalized plans [LETOV86]. These are stereotypic action sequences in a program which are not necessarily contiguous segments of code. However, because of the absence of a standard or a widely-accepted set of criteria for identification of plans across programming languages and application areas, only a brief abstract treatment of non-contiguous "program parts" is mentioned in the next section.

3.1.3 Theoretical Perspective and Limitations

Now that we have defined the possible ways in which the objects in a program can be identified, we need to find a way to relate these objects to program reusability. But before we do that, we need to establish a basis for comparison. Assume that there is a program, S, consisting of n_m modules, n_c components, and n_s statements. Modules are composed of components and components are composed of statements. Consequently, in general, to keep the potential for reuse high, S can be considered as a set of modules plus some components and even some individual statements.

We can study the reusability of a program, S, by assuming that a new program, S', is to be written and that we can identify a set of existing program parts (statements, components, and/or modules) that can be included in S' thus saving the effort of writing them from scratch. In this manner, we can model or simulate the alternative to writing new code which is reusing an "existing" portion of code as it can be obtained "off the shelf."

The basic case can be established by removing the barriers from all modules and components in S, i.e., allowing all of the statements to be visible to one another. In this case, we attempt to simulate a worst case scenario in which the program has been developed from scratch. A case where a programmer was asked to write the entire

program without the option of reusing any modules or components.

At the other extreme, we can model an optimum reuse case where the programmer was asked only to write the main program statements and had the opportunity of reusing all the other independent modules and/or components that exist in the program.

Now that we have described the boundary cases for our study of reusability, we can describe a third case which seems to fit inside our reusability spectrum. A case in which the programmer is asked to developed the main program statements along with the statements for some of the components anew, and has the option of using new modules that combine some of the original components. This generic case brings out the problem of determining which of the statements, components, and/or modules are the best candidates to be "reused".

In general, we have a partition, π, on a program, S, defined as a collection of disjoint and nonempty subsets of statements in S whose union is S, i.e., $\pi = (B_\alpha)$, $\alpha \in I$ where I is an indexing set, such that $B_\alpha \neq \Phi$, for all $\alpha \in I$, $B_\alpha \cap B_\beta = \Phi$ for $\alpha \neq \beta$, and $U\ (B_\alpha) = S$ where $\alpha \in I$. We refer to the sets in π as blocks of π [HARTM66]. For example, if S is a set consisting of 4 elements, S = {1,2,3,4}, we can see that π can be written in 8 different ways if the element sequence is to be preserved, i.e., {{1},{2},{3},{4}}, {{1,2},{3},{4}}, {{1},{2,3},{4}}, {{1},{2},{3,4}},

{{1,2},{3,4}}, {{1,2,3},{4}}, {{1},{2,3,4}}, and {{1,2,3,4}}. Thus, as the size of S increases, the number of ways in which π can be written increases much faster.

To have an intuitive appreciation of the above-mentioned increase, consider the problem of the number of partitions of an integer, p(n) [GROSS84 and HALL67]. p(n) represents the total number of ways in which an integer, n > 0, can be represented as a sum of positive integers if he instances that differ only by the order of the summands are not considered different. Each such representation is called a partition of n. For example, p(4) = 5 (i.e., 1+1+1+1, 1+2+1, 1+3, 2+2, and 4) while p(25) = 1958 [HALL67]. Consequently, in a program with a large number of statements, all the possible ways in which the statements can be considered becomes prohibitively time consuming.

3.1.4 Coupling and Cohesion

To reduce the number of possible combinations of objects, we use the notions of coupling and cohesion. Coupling, as defined by Stevens, Myers, and Constantine [STEVE74], is a measure of the strength of the association established by a connection from, in this case, one object to another. Cohesion is defined by Stevens et al. and restated by Booch [BOOCH86] as how tightly bound or related the internal elements within an object are to one another.

Stevens et al. [STEVE74] also state that "the fewer and simpler the connections between objects, the easier it is to

understand each object without reference to other objects." The complexity of a system is affected not only by the number of connections but by the degree to which each connection couples two objects, making them interdependent rather than independent. Thus coupling is reduced when the relationships between objects are minimized or, in Chanon's terms, when the number of assumptions that the objects make is minimized.

One way to minimize coupling is to maximize the relationships among elements within the same object, i.e., obtaining the objects that display the highest cohesiveness [STEVE74]. Consequently, we can reduce the number of possible statement combinations by combining into objects the statements that possess the highest cohesiveness, thus resulting in the lowest degree of coupling. Similarly, we can combine into modules the components that possess the highest cohesiveness.

Based on the aforementioned discussion on coupling, we can now identify some realistic combinations of the statements, components, and/or modules in the original program that potentially can be selected for reuse to allow us to compute the entropy loading and control structure entropy of the resulting program consisting of the reused as well as original parts.

3.2 Carrying out the Experiment

3.2.1 Quantifying Software Reuse

The main objective of the experiment is to determine if a relation exists between software reusability and the information theory based metrics: entropy loading and control structure entropy. The major problem is finding a mechanism that can help us quantify the notion of software reuse.

As mentioned in Chapter II, one of the benefits of software reusability is the reduction in the amount of software that needs to be written when software is available for reuse. The amount of software reused can be correlated with the amount of code that needs to be developed from scratch for a given program. The more code that is available for reuse, the less new code is needed for the new program. Thus, intuitively, an inverse relation exists between software reused and the amount of software needed to be developed anew.

For this study, the lines of code metric[2] was used to quantify the amount of code that needs to be developed from scratch. The lines of code metric was not applied to the "reused" segments of code since they are not considered part of the effort of writing the new program.

---

[2]A line of code is defined by Conte et al. [CONTE86] as "any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line."

3.2.2 Experiment Operation

The next two subsections explain the software tools that were developed and used to collect the measurements.

3.2.2.1 Programs Developed to Collect the Data.

Because pre-written software packages that compute entropy loading and lines of code metrics were not available, software tools to collect the measurements had to be developed. A total of three programs in C [KERN78] were developed on a VAX 11/785 running ULTRIX[3] (see Appendix A for program listings). The first program, called ent_lo5.c, was developed to compute the entropy loading of a collection of objects in a given program based on the set of assumptions made by the objects as demonstrated by Chanon [CHANO73]. In addition to the entropy loading, this program also computes the average object entropy, system entropy (H(S)), and the ratio of entropy loading to the total number of objects. No program was developed to extract the assumptions made by the objects in the programs. All assumptions were manually extracted from the programs based on Chanon's work [CHANO73] and are listed in Appendix D.

The other two programs, called loc_c2.c and loc_ada1.c, were developed to compute the lines of code metric for the C and Ada language programs used in the experiment. Both programs compute the lines of code metric as defined by

---

[3]VAX and ULTRIX are Trademarks of Digital Equipment Corporation.

Conte et al. [CONTE86] but go one step further. The lines of code was partitioned into three categories for the C programs, namely, declaration lines, non-declaration lines, and brace lines ("{" or "}" and no C statements). Ada lines of code were partitioned into declaration and non-declaration lines only. Declaration lines were extracted to investigate if there is any relation between them and the information contained in the objects (i.e., average object entropy and system entropy). Additionally, a relation is expected between the number of brace lines in C programs and MIN since they are expected to correlate well with the depth of nesting in programs.

3.2.2.2 Data Collection. A total of three versions for each of the programs included in the study (see Section 3.2.3) were considered. These versions follow the guidelines for reuse established earlier as follows: version 1, optimum reuse; version 2, intermediate reuse; and version 3, no reuse. All metrics were applied to all three versions of the programs.

The following convention was used to identify the objects in the programs under study. All object numbers are of the form A.4, C.0, or FHI.0. A.4 corresponds to the fourth statement in function or procedure A, C.0 corresponds to a component which identifies function or procedure C, and FHI.0 corresponds to a module composed of components F, H, and I.

All measurements, with the exception of the control structure entropy, were collected on the VAX 11/785 running ULTRIX. Control structure entropy, Z_n, was the only metric that was not computed using a program. This metric was computed using the Maximal Intersect Number charts in Appendix F and the equation

$$Z_n = MIN - 1$$

where MIN is the Maximal Intersect Number determined from the charts as demonstrated by Chen [CHEN78]. The object numbers are shown inside the IF symbols in the charts in Appendix F.

3.2.3 Programs Studied in the Experiment

A total of six programs obtained from the open literature were studied in this experiment (see TABLE I and Appendixes B and C). Three of the programs were written in

TABLE I

TESTBED PROGRAM SOURCES

Name	Language	Application	Source
fastfind	C	string processing	[MILLE87]
mail	C	database	[SCHIL87]
editor	C	string processing	[SCHIL87]
int_list	Ada	string processing	[SHUMA89]
calc	Ada	numeric	[MOHNK86]
address	Ada	database	[MOHNK86]

C [KERNI78] and the other three in Ada [ADA83]. A noticeable difference between the C and Ada programs studied was the number of compilation units. All C programs were compiled as single units, while all Ada programs had two or more compilation units per program.

The following subsections describe the main purpose of each of the programs along with a description of the objects used in the intermediate reuse case. The set of objects used in the intermediate reuse case are chosen to be a mixture of the three types of objects available.

3.2.3.1 C Language Programs. The first C language program is a program called fastfind. The main purpose of this program is to search through one or more files containing ASCII text looking for a match for a character string given as input. The input string may consist of any sequence of up to eighty characters including spaces. The output consists of the entire text line in which a match was found for the input string preceded by the name of the file to which the text line belongs if more than one file was specified.

The fastfind program consists of a main program and six functions (See Appendix E, Figure 2). Out of the six functions, two functions (fill_buffer and print_line) were combined into a module called module_1 in the intermediate reuse case. These functions were combined because of the logical binding [STEVE74] exhibited between them. Function

scan was not considered to be a candidate for reuse so all of its statements were visible to the rest of the objects. The remaining functions were "reused" as components.

The second C language program is a program called mail. The main purpose of this program is to create and update a personal mailing list database with a total of five fields per record (name, street, city, state, and zip code). There are a total of six options available on the database. The user can enter new records, delete existing records, list all records, search for a particular record, save all records to a file, and load an existing database from a file. The primary field in the database is the field name.

The mail program consists of a main program and eleven functions (See Appendix E, Figure 3). Out of the eleven functions, three functions (enter, inputs, and dls_store) were combined into a module called module_1 in the intermediate reuse case. These functions were combined because of the temporal binding [STEVE74] exhibited between them since all of these functions are executed at one time. Functions list, search, and delete were the functions not selected as candidates for reuse in this case. The remaining functions were "reused" as components.

The last C language program is a program called editor. This program simulates a simple text editor with very limited capabilities. There are five options available to the user. The user can enter one or more lines of text in the same operation beginning at the specified line number or

delete a line of text based on the line number. He/she can also list the contents of the text file, save all text into a file, and retrieve an existing text file from a file. All lines are indexed by the line numbers in each line.

The editor program consists of a main program and nine functions (See Appendix E, Figure 4). Out of the nine functions, two functions (patchup and find) were combined into a module called module_1 in the intermediate reuse case. These functions were combined because of the logical binding [STEVE74] exhibited between them. Functions enter and delete were the functions that were not selected for reuse in this case. The remaining functions were considered to be objects that are candidates for reuse as components.

3.2.3.2 Ada Language Programs. The first Ada language program studied is a program called int_list. The main purpose of this program is to illustrate the concepts of dynamic allocation and garbage collection. Memory is dynamically allocated for a list of numbers accepted from the keyboard as the numbers are added both to the beginning and end of the list. Subsequently, garbage collection is used to reclaim memory allocated as numbers are deleted from the beginning and the end of the list. The remaining numbers are then printed to demonstrate that the first and last numbers were deleted.

The int_list program consists of a main program called exercise_20_1, seven procedures, and two functions (see

Appendix E, Figure 5). Out of the seven procedures, three procedures (insert_at_head, insert_at_tail, and alloc) were combined into a module called module_1 in the intermediate reuse case. These procedures exhibit communicational binding [STEVE74] as they reference the same input data and are related in time. Procedures delete_tail and delete_head were not selected as candidates for reuse. The remaining functions and procedures were considered for reuse as components.

The second Ada language program used in the study is a program called calc. This program simulates a four function calculator using Reverse Polish Notation (RPN). In RPN, the equal sign is never used. All operators precede the operation and when the operation symbol is entered, the result is displayed. Consequently, no parentheses are needed either. One of the good points about this program, is that it accepts numeric input in any of the common notations, including integers, signed integers, floating point fractions starting with a decimal point, signed floating point fractions, and any of the above followed by an exponent. A stack of ten entries is also provided to store the last ten operands entered.

The calc program consists of a main program called calculate_2 and seven procedures (see Appendix E, Figure 6). Out of the seven procedures, three procedures (push, pop, and clear) were combined into a module called module_1 in the intermediate reuse case. These procedures were the only

ones that exhibited some binding among themselves although it was only of a temporal type [STEVE74]. Procedure operate was not selected for reuse in this case hence all of its statements were visible to the rest of the objects. The remaining procedures were considered for reuse as components.

The last Ada language program included in the experiment is a program called address. This programs provides access to an address book database. The concept is similar to the C language program mail, although the implementation is completely different. Fields are provided to accommodate name, street address, city, state, zip code, and telephone number. The field name is used as the only key in the database.

A separate index file based on name entries was maintained. To speed up search operations, binary search was provided as the search mechanism in contrast with sequential search used in mail. The operations provided include database initialization, new address insertion, address deletion, address modification, and address search based using the name as the key.

The address program consists of a main program called address_book and twelve procedures (see Appendix E, Figure 7). Out of the twelve procedures, two of the procedures (alter_data and alter_field) were combined into one module called module_1 in the intermediate case of reuse. These two procedures exhibit communicational binding [STEVE74] as

they refer to the same input data obtained from the keyboard and are related in time, i.e., executed at data alteration time. Procedures delete and insert were not reused in this case. On the other hand, all other procedures were considered for reuse as components.

CHAPTER IV

ANALYSIS OF MEASUREMENTS

4.1 Description of the Analysis

All data analysis was done on a VAX 8550 running VMS[1] using SAS [SAS85a,b]. Standard statistical methods described by Conte et al. [CONTE86] were used.

The sample sizes for this study on software reusability using Ada and C programs were not arrived at statistically; rather, three correct programs written in each language found in the open literature were used. The three Ada programs have 137, 168, and 493 nonblank, noncomment lines of code and the three C programs have 138, 272, and 286 nonblank, noncomment lines of code. Each of the three versions for each of the programs were considered as separate cases (optimum, intermediate, and no reuse cases).

Pearson product-moment correlations [SAS85a,b] were computed between the lines of code metric and the control structure entropy metric, and between the lines of code metric and the entropy loading metric. Use of this correlation method requires that the measurements be parametric [CONTE86]. The measurements should be

[1]VMS is a Trademark of Digital Equipment Corporation.

independent, drawn from normally distributed populations, the populations should have nearly the same standard deviations, and all measurements should be in at least the same interval scale (meaning that the data have meaningful differences and can be ranked and categorized).

Correlations were also computed within the lines of code class and within the entropy loading class. The lines of code class is composed of lines of code (LOC), declaration lines (Dec), non-declaration lines (NDe), Brace lines (Bra, only used with C programs), comment lines (Com), and blank lines (Bla). The entropy loading class is composed of the object total (Obj Tot), average object entropy, system entropy loading to object total ratio (SEL to OT Ratio), system entropy, and system entropy loading. See TABLE II for a list of the measurements obtained. TABLE III contains some of the correlations between the metrics. No correlations were computed between the entropy loading metric and the control structure entropy as these correlations were deemed outside of the scope of the experiment. A complete list of the correlations is provided in Appendix G.

Correlations using natural logarithm transformations of the measurements were computed. The next two sections analyze the results obtained from the correlations of the metrics.

TABLE II

METRICS EVALUATED FOR PROGRAMS

Program Name	Lines of Code							Con Str Ent	Obj Tot	Average Object Entropy	SEL to OT Ratio	System Entropy	System Entropy Loading
	LOC	Dec	NDe	Bra	Com	Bla	Tot						
fastfind1	63	15	36	12	42	8	113	5	32	0.78965	0.63535	4.93749	20.33123
fastfind2	83	16	49	18	50	11	144	8	39	0.68940	0.55519	5.23408	21.65248
fastfind3	138	20	84	34	62	24	224	13	60	0.59903	0.50359	5.72652	30.21552
mail1	38	6	26	6	10	5	53	7	19	1.71870	1.49513	4.24793	28.40739
mail2	89	9	60	20	11	14	114	13	38	1.35084	1.21690	5.09000	46.24208
mail3	272	20	184	68	22	37	331	31	109	1.03882	0.98286	6.09940	107.13164
editor1	50	6	38	6	10	4	64	6	27	1.04682	0.87346	4.68081	23.58340
editor2	110	10	79	21	12	10	132	13	54	1.01528	0.91283	5.53210	49.29293
editor3	286	23	197	66	22	31	339	26	131	0.94514	0.89626	6.40318	117.41041
int_list1	29	1	28	---	10	5	44	1	17	1.70926	1.48959	3.73452	25.32298
int_list2	66	6	60	---	21	10	97	2	28	1.22103	1.05954	4.52164	29.66723
int_list3	137	11	126	---	23	17	177	8	59	0.96118	0.87409	5.13800	51.57139
calc1	47	5	42	---	11	3	61	3	17	0.87472	0.63428	4.08746	10.78283
calc2	67	6	61	---	11	3	81	16	27	1.27941	1.11427	4.45859	30.08542
calc3	168	12	146	---	33	10	211	24	57	0.67578	0.57654	5.65664	32.86275
address1	104	27	77	---	20	8	132	8	39	1.41669	1.28643	5.08023	50.17061
address2	199	37	162	---	47	19	265	13	77	1.05323	0.97696	5.87266	75.22577
address3	493	58	435	---	127	59	679	36	181	0.52929	0.49125	6.88565	88.91667

TABLE III

CORRELATIONS BETWEEN METRICS

(Ada)	LOC	Dec	NDe	Com	Bla	Tot
ConStrEnt	.8551	.8385	.8467	.6925	.5225	.8136

(Ada)	LOC	Dec	NDe	Com	Bla	Tot
ObjTot	.9860	.8693	.9886	.9616	.9279	.9921
AveObjEnt	-.7692	-.6495	-.7934	-.7427	-.6034	-.7619
SELtoOTRat	-.6368	-.5398	-.6619	-.6117	-.4529	-.6257
SysEntr	.9852	.9346	.9745	.9505	.8898	.9856
SysEntLoa	.8095	.7265	.7978	.7949	.8489	.8234

(C)	LOC	Dec	NDe	Bra	Com	Bla	Tot
ConStrEnt	.9284	.5944	.9510	.9101	.0077	.8900	.8395

(C)	LOC	Dec	NDe	Bra	Com	Bla	Tot
ObjTot	.9947	.8142	.9897	.2896	.9670	.9168	.9637
AveObjEnt	-.3292	-.6901	-.2442	-.8891	-.3626	-.3524	-.4985
SELtoOTRat	-.1778	-.5906	-.0887	-.8856	-.2169	-.2141	-.3615
SysEntr	.9785	.8549	.9609	.3942	.9656	.9142	.9732
SysEntLoa	.8537	.4589	.8969	-.2007	.8061	.7597	.7251

4.2 Entropy Loading Analysis

Entropy loading, as described in Chapter II, is a measure of the information shared among collections of objects as opposed to the information used inside each collection [CHAN073]. Consequently, we can expect a higher entropy loading as the number of collections making assumptions increases (considering each object as a collection only containing itself) since the total amount of information shared increases. This assumption was verified by a strong, positive, and significant correlation (significance of 0.004 or less) between entropy loading and the total number of objects for both C and Ada programs.

The increase in the number of collections was represented by an increase in the amount of code (quantified by the lines of code metric) that needs to be written when the opportunity for reuse is smaller. As expected, a strong, positive, and significant correlation (significance of 0.0082 or less) was found between the lines of code and the total number of objects, and between lines of code and entropy loading for both C and Ada programs. In other words, the larger the amount of code needed anew (smaller reuse), the larger the number of objects and consequently, the higher the entropy loading.

Overall, strong, positive, and significant correlations (significance of 0.027 or less) were found between the total number of objects, system entropy, and entropy loading on

one side and most of the lines of code metrics class for both C and Ada programs on the other side.

Additionally, strong, negative, and significant correlations (significance of 0.02 or less) exist between the average object entropy measure and most of the measures in the lines of code class for the Ada programs. The average object entropy is an indicator of the information contained inside objects.

4.3 Control Structure Entropy Analysis

While entropy loading is used as a measure of information among collections of objects, control structure entropy, defined in Chapter II, is used as a measure of the complexity of a program [CHEN78]. Consequently, a higher control structure entropy is expected as the perceived complexity of the program increases.

One of the assumptions that we are trying to validate (or at least provide support for) in this study is that the more code that is available for reuse, the less complex the resulting new program tends to be. This is expected because the internal complexity of the reusable modules is not seen by the programmer when the new program is being developed since the modules already exist and either have been previously understood or can be understood in one chunk.

Strong, positive, significant correlations (significance of 0.0033 or less) exist between the control structure entropy metric and the lines of code metric for both C and

Ada programs. This should be expected as the lines of code metric is used to quantify the amount of software that needs to be developed anew, validating our assumption. In other words, the less code available for reuse, the more complex the new program tends to be.

The relation between MIN and the brace lines measure discussed in Section 3.2.2.1 was verified by a strong, positive, significant correlation (significance of 0.0007) between them.

CHAPTER V

SUMMARY, CONCLUSIONS, AND FUTURE WORK

The main theme of this thesis was to theoretically explore the relationships between software reusability and two information theory based metrics: entropy loading and control structure entropy. A survey of the open literature indicated that previous work in this area had not addressed the idea of quantifying software reuse with this type of metrics.

Entropy loading was found to be inversely proportional to the amount of reuse present in the programs. Entropy loading was always smaller in the optimum reuse cases. This was corroborated by strong correlations found between entropy loading and the size of the resulting new program, measured by the lines of code metric. Consequently, entropy loading can presumably provide a mechanism for selecting the optimum reuse case among different possibilities for reuse.

Control structure entropy, a measure of the complexity of a program, was also found to be a good indicator of reuse. The optimum reuse case (higher reuse) was always found to be the one with the lowest control structure entropy. Strong correlations exist between control

structure entropy and the size of the resulting new program, measured by the lines of code metric.

In conclusion, there seems to be a relation between entropy loading and software reuse, and between control structure entropy and software reuse. But, care should be taken not to make any irrational generalizations since this study was not a controlled experiment and the sample sizes were not arrived at statistically. The intent of this study was only to determine if a possible relation between the metrics and software reuse existed.

Suggestions for future work include conducting a similar, but controlled and larger scale experiment to test the hypothesis that the notion of software reuse can be quantified. Perhaps, adding other software metrics to the ones used in this study and/or adding other programming languages would provide more insight. Possible candidates include other information theory based metrics such as residual complexity [SAMAD88] and other metrics such as software science metrics [HALST79].

Other work might also include software reuse instances where some of the objects overlap (i.e., there is a certain degree of harmless overkill involved in the objects that are being reused). In such a case, entropy loading can not be applied but other measures can be applied (e.g., [SAMAD88 and SCHUT77]).

Finally, automated tools can be developed to determine assumptions made by objects in the calculation of the

entropy loading metric. This task is considered the most time consuming in the application of the entropy loading metric.

REFERENCES

[ADA83] Reference Manual for the Ada Programming Language, United States Department of Defense, ANSI/MIL-STD-1815A, January 1983.

[ALEXA64] Alexander, Christopher, Notes on the Synthesis of Form, Harvard University Press, Cambridge, Mass., 1964.

[BERLI80] Berlinger, Eli, "An Information Theory Based Complexity Measure," Proceedings of the 1980 ACM National Computer Science Conference, Arlington, VA, AFIPS Press, pp. 773-779.

[BIGGE87] Biggerstaff, Ted and Richter, Charles, "Reusability: Framework, Assessment, and Directions," IEEE Software, March 1987, pp. 41-49.

[BOOCH86] Booch, Grady, Software Engineering with Ada, The Benjamin/Cummings Publishing Company Inc., Second Edition, 1986.

[CHANO73] Chanon, Robert N., "On a Measure of Program Structure," Ph.D. Dissertation, Department of Computer Sciences, Carnegie-Mellon University, Pittsburgh, PA, November 1973.

[CHEAT83] Cheatham, Jr., T. E., "Reusability Through Program Transformations," Proceedings ITT Workshop on Reusability in Programming, September 7-9, 1983, pp. 122-128.

[CHEN78] Chen, Edward T., "Program Complexity and Programmer Productivity," IEEE Transactions on Software Engineering, Vol. SE-4, No. 3, May 1978, pp. 187-194.

[CHENG84] Cheng, Thomas T., Lock, Evan D., and Prywes, Noah S., "Use of Very High Level Languages and Program Generation by Management Professionals," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 552-563.

[CONTE86] Conte, S. D., Dunsmore, H. E., and Shen, V. Y., Software Engineering Metrics and Models, The Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1986.

[DIJKS72] Dijkstra, E. W., "Notes on Structured Programming," Structured Programming, Academic Press, New York, 1972.

[FREEM83] Freeman, Peter, "Reusable Software Engineering: Concepts and Research Directions," Proceedings of the ITT Workshop on Reusability in Programming, September 7-9, 1983, pp. 2-15.

[GOGUE86] Goguen, Joseph A., "Reusing and Interconnecting Software Components," IEEE Computer, Vol. 19, February 1986, pp. 16-28.

[GROSS84] Grosswald, Emil, Topics from the Theory of Numbers, Birkhäuser, 1984.

[HALL67] Hall Jr., Marshall, Combinatorial Theory, Blaisdell Publishing Co., 1967.

[HALST79] Halstead, M. H., "Advances in Software Science," Advances in Computers, (Yovits, ed.), Vol. 18, Academic Press, New York, 1979, pp. 119-172.

[HARTM66] Hartmanis, J. and Stearns, R. E., Algebraic Structure Theory of Sequential Machines, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1966.

[KAISE87] Kaiser, Gail E. and Garlan, David, "Systems from Reusable Building Blocks," IEEE Software, July 1987, pp. 17-24.

[KERNI78] Kernighan, Brian W. and Ritchie, Dennis M., The C Programming Language, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[KERNI84] Kernighan, Brian W., "The UNIX System and Software Reusability," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 513-518.

[LEE87] Lee, Tony T., "An Information-Theoric Analysis of Relational Databases-Part I: Data Dependencies and Information Metric," IEEE Transactions on Software Engineering, Vol. SE-13, No. 10, October 1987, pp. 1049-1061.

[LETOV86] Letovsky, Stanley and Soloway, Elliot, "Delocalized Plans and Program Comprehension," IEEE Computer, May 1986, pp. 41-49.

[LUBAR86a] Lubars, Mitchell D., "Code Reusability in the Large Versus Code Reusability in the Small," ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 1, January 1986, pp. 21-27.

[LUBAR86b] Lubars, Mitchell D., "Affording Higher Reliability Through Software Reusability," ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 5, October 1986, pp. 39-42.

[MATSU84] Matsumoto, Yoshihiro, "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 502-513.

[MCCAB76] McCabe, J., "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 308-320.

[MILLE87] Miller, Webb, A Software Tools Sampler, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

[MOHAN79] Mohanty, Siba N., "Models and Measurements for Quality Assessment of Software," ACM Computing Surveys, Vol. 11, No. 3, September 1979, pp. 251-275.

[MOHAN81] Mohanty, Siba N., "Entropy Metrics for Software Design Evaluation," The Journal of Systems and Software 2, 1981, pp. 39-46.

[MOHNK86] Mohnkern, Gerald L. and Mohnkern, Beverly, Applied Ada, Tab Professional and Reference Books, 1986.

[PARNA71] Parnas, D. L., "Information Distribution Aspects of Design Methodology," Proceedings of the IFIP Congress, Ljubljana, Yugoslavia, 1971.

[PRIET87] Prieto-Díaz, Rubén and Freeman, Peter, "Classifying Software for Reusability," IEEE Software, January 1987, pp. 6-16.

[SAMAD88] Samadzadeh, Mansur H. and Edwards, William R., Jr., "A Classification Model of Comprehension," Twenty First Annual Hawaii International Conference on System Sciences (HICSS21), Hawaii, 1988.

[SAS85a] SAS User's Guide: Basics, Version 5 Edition, SAS Institute Inc., Box 8000, Cary, NC 27511, 1985.

[SAS85b] SAS User's Guide: Statistics, Version 5 Edition, SAS Institute Inc., Box 8000, Cary, NC 27511, 1985.

[SCHIL87] Schildt, Herbert, Advanced Turbo C, Osborne McGraw-Hill, New York, NY, 1987.

[SCHUT77] Schütt, Dieter, "On a Hypergraph Oriented Measure for Applied Computer Science," Proceedings of COMPCON, Washington, D.C., Fall 1977, pp. 295-296.

[SHANN64] Shannon, Claude E. and Weaver, Warren, The Mathematical Theory of Communication, The University of Illinois Press, Urbana, Ill., 1964.

[SHOOM83] Shooman, Martin L., Software Engineering: Design, Reliability, and Management, McGraw-Hill Book Company, New York, 1983.

[SHUMA89] Shumate, Ken, Understanding Ada with Abstract Data Types, John Wiley and Sons, Inc., New York, NY, Second Edition, 1989.

[SIMON62] Simon, H. A., "The Architecture of Complexity," Proceedings of the American Philosophical Society, 106, December 1962, pp. 467-482.

[SOMME89] Sommerville, Ian, Software Engineering, Addison-Wesley Publishing Co., Third Edition, 1989.

[STEVE74] Stevens, W. P., Myers, G. J., and Constantine, L. L., "Structured Design," IBM Systems Journal, Vol. 2, 1974, pp. 115-139.

[TRACZ88] Tracz, Will, "Software Reuse Maxims," ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 5, October 1988, pp. 28-31.

[VANEM70] van Emden, M. H., "Hierarchical Decomposition of Complexity," Machine Intelligence 5, 1970, pp. 361-380.

[WEGNE83] Wegner, Peter, "Varieties of Reusability," Proceedings of the ITT Workshop on Reusability in Programming, September 7-9, 1983, pp. 30-44.

SELECTED BIBLIOGRAPHY

1. Belady, L. A. and Evangelisti, C. J., "System Partitioning and Its Measure," The Journal of Systems and Software 2, 1981, pp. 23-29.

2. Brown, P. J., Software Portability, Cambridge University Press, 1977.

3. Chandersekaran, C. S. and Perriens, M. P., "Towards an Assessment of Software Reusability," Proceedings of the ITT Workshop on Reusability in Programming, September 7-9, 1983, pp. 179-182.

4. Hellerman, Leo, "A Measure of Computational Work," IEEE Transactions on Computers, Vol. C-21, No. 5, May 1972, pp. 439-446.

APPENDIXES

APPENDIX A

PROGRAMS USED TO COMPUTE THE METRICS

```
/*********************************************************************
*           E n t r o p y   L o a d i n g   M e t r i c             *
*********************************************************************
*                                                                   *
*           File:  ent_lo5.c                                        *
*           Author:  William R. Torres                              *
*           Date:  90/01/25                                         *
*           Class:  COMSC 5000 - Thesis                             *
*           Adviser:  Dr. Mansur Samadzadeh                         *
*                                                                   *
*********************************************************************
*                                                                   *
*   This program computes the entropy loading metric as presented   *
*   by Chanon [CHAN074].  The input for this program consists of the*
*   name of the file that contains the input data for the program.  *
*   The first line of the input file must contain the total number  *
*   of subsystems, which in our case is always the same as the      *
*   number of objects, the total number of objects, and the total   *
*   number of assumptions.                                          *
*                                                                   *
*   The remaining input lines are divided into two groups.  Each    *
*   line in the first group corresponds to each of the objects in   *
*   the program under study.  Each line contains the object's name  *
*   (i.e., A.1, C.2), the total number of assumptions the object    *
*   makes, and the numbers of the assumptions the object makes      *
*   (i.e., 2 4 5 when the object makes assumptions 2, 4, and 5).    *
*   Each line in the second group corresponds to each subsystem in  *
*   the program under study.  Each of these lines contain the       *
*   total number of objects in the subsystem and the numbers that   *
*   identify the objects in the subsystem.                          *
*                                                                   *
*   The output of this program is divided into two groups.  In the  *
*   first group, the program produces a line with the name of the   *
*   object and the entropy for each of the objects in the program.  *
*   The next output group produces the total number of objects, the *
*   average object's entropy (sum of entropies divided by total     *
*   number of objects), the system entropy, the system entropy      *
*   loading, and the ratio of system entropy loading to total       *
*   number of objects.                                              *
*                                                                   *
*********************************************************************/

#include <stdio.h>
#include <math.h>
#define OBJECTS 190
#define ASSUMPTIONS 170
#define SUBSYSTEMS 190

main()
  {
  int i, j, k, m, n, p;
  double entropy[OBJECTS + 1],     /*subsystem entropy calculation*/
         entropy_loading,          /*system entropy loading*/
         double_sum,               /*temporary double variable*/
         temp_double;              /*temporary double variable*/
```

```
int OAT[OBJECTS][ASSUMPTIONS],  /*Object/Assumption Table*/
    subsys_cnt,                 /*subsystem count*/
    subsys_size[SUBSYSTEMS],    /*No. of Objects in Subsys.*/
    sub_system[SUBSYSTEMS][OBJECTS],  /*Subsys. Objects*/
    obj_cnt,                    /*number of Objects in OAT*/
    asmp_cnt,                   /*assumption count in OAT*/
    asmp_no,                    /*Number of assmp. for Object*/
    asmp_col,                   /*columns with assumptions*/
    occurrences[OBJECTS],       /*submatrix element occurrences*/
    NODO,                       /*Number Of Distinct Occurrences*/
    sub_matrix[OBJECTS][ASSUMPTIONS],  /*OAT temporary submatrix*/
    submatrix_size,             /*submatrix row size*/
    columns[ASSUMPTIONS][2],    /*columns where assmp. are present*/
    columns_head,               /*columns list head*/
    rem_obj[OBJECTS];           /*remaining rows not compared yet*/
char file_buffer[25];           /*data filename buffer*/
struct {
  char name[10]; } obj[OBJECTS]; /*object names*/
FILE *datafp,                   /*data file pointer*/
     *outputfp;                 /*output file pointer*/
double loga_n();
double power();

strcpy(file_buffer, "");
while(strcmp(file_buffer, "quit") != 0)
  {
  printf("\nEnter data filename or quit to stop program\n==>> ");
  scanf("%s", file_buffer);  /*get data filename*/
  if(strcmp(file_buffer, "quit") != 0)
    {
    if((datafp = fopen(file_buffer, "r")) == NULL)  /*open file*/
      printf("Error opening file %s\n", file_buffer);
    else  /*file opened for reading*/
      {
      outputfp = fopen("ent_load_out", "w");  /*open output file*/
      /*Read No. of Subsystems, Objects, and Assumptions*/
      fscanf(datafp, "%d %d %d", &subsys_cnt, &obj_cnt, &asmp_cnt);
      for(i = 0;i < obj_cnt; i++)  /*initialize OAT to zero*/
        for(j = 0;j < asmp_cnt; j++)
          OAT[i][j] = 0;
      for(i = 0;i < obj_cnt; i++)  /*assumption input loop*/
        {
        fscanf(datafp, "%s", obj[i].name);  /*read object's name*/
        fscanf(datafp, "%d", &asmp_no); /*read number of assumptions*/
        for(j = 0;j < asmp_no; j++)  /*read assumptions*/
          {
          fscanf(datafp, "%d", &asmp_col);
          OAT[i][asmp_col - 1] = 1;  /*assign assumptions*/
          }  /*end for*/
        }  /*end for*/
      for(i = 0;i < subsys_cnt; i++)  /*subsystem input loop*/
        {
        fscanf(datafp, "%d", &subsys_size[i]);  /*no. of objects*/
        for(j = 0;j < subsys_size[i]; j++)  /*read object numbers*/
          {
```

```
    fscanf(datafp, "%d", &sub_system[i][j]);
    sub_system[i][j]--;  /*adjust object number*/
    }  /*end for*/
  }  /*end for*/
printf("\nObject Name      Entropy (H(X))\n");
printf("================================\n");
for(i = 0;i < subsys_cnt; i++)  /*subsystem loop*/
  {
  submatrix_size = 0;
  columns_head = 0;  /*initialize list*/
  for(n = 0;n < subsys_size[i]; n++)
    {  /*Find Assmp. made in the subsys. objects*/
    for(j = 0;j < asmp_cnt; j++)
      {  /*obtain subsystem submatrix from OAT*/
      if(OAT[sub_system[i][n]][j] == 1) /*1 pres. in the col.*/
        {
        if(submatrix_size == 0)  /*empty list*/
          {
          columns[submatrix_size][0] = j;  /*save col. no.*/
          columns[submatrix_size++][1] = -1;  /*end*/
          }  /*end if*/
        else  /*list is not empty*/
          {
          for(k = columns_head;k > -1;p = k,k = columns[k][1])
            {  /*search list to insert column number*/
            if(j < columns[k][0])  /*insert before present*/
              {
              columns[submatrix_size][1] = k; /*link to pres.*/
              columns[submatrix_size++][0] = j; /*save col. #*/
              if(k == columns_head) /*pres. was first*/
                columns_head = submatrix_size - 1; /*new first*/
              else  /*pres was not first, link to previous*/
                columns[p][1] = submatrix_size - 1;
              break;  /*end search*/
              }  /*end if*/
            else if(j == columns[k][0])  /*repeated column*/
              break;  /*stop search*/
            }  /*end for*/
          if((j > columns[p][0]) && (k == -1))
            {  /*place at the end of the list*/
            columns[submatrix_size][0] = j;  /*save col. no.*/
            columns[submatrix_size++][1] = -1;  /*end*/
            columns[p][1] = submatrix_size - 1; /*link back*/
            }  /*end if*/
          }  /*end else*/
        }  /*end if*/
      }  /*end for*/
    }  /*end for*/
  for(j = 0;j < obj_cnt; j++)  /*form submatrix*/
    for(k = columns_head,m = 0;k > -1; m++,k = columns[k][1])
      sub_matrix[j][m] = OAT[j][columns[k][0]];
  for(j = 0;j < obj_cnt; j++)
    rem_obj[j] = 0;  /*mark rows as not compared yet*/
  for(j = 0,NODO = 0;(j < (obj_cnt - 1)); j++)
    {  /*jth row is compared with mth row*/
```

```
      m = j + 1;
      if(rem_obj[j] != 1)  /*row not previously compared*/
        {
        rem_obj[j] = 1;  /*remove object from submatrix*/
        occurrences[NODO++] = 1;  /*initial occurrence*/
        while(m < obj_cnt)
          {  /*m is the following row index*/
          if(rem_obj[m] != 1)  /*row not matched before*/
            {
            for(k = 0;((sub_matrix[j][k] == sub_matrix[m][k]) &&
              (k < submatrix_size)); k++);  /*compare rows*/
            if(k == submatrix_size)  /*matching rows*/
              {
              occurrences[NODO - 1]++;  /*increase count*/
              rem_obj[m] = 1;  /*remove object from submatrix*/
              }  /*end if*/
            }  /*end if*/
          m++;  /*get ready for next row*/
          }  /*end while*/
        }  /*end if*/
      }  /*end for*/
    if((j == (obj_cnt - 1)) && (rem_obj[j] == 0))
      occurrences[NODO++] = 1;  /*last row is unique*/
    /*Entropy function computation for the subsystems*/
    entropy[i] = (double) obj_cnt;
    entropy[i] = loga_n(entropy[i]);  /* log |X| */
    double_sum = 0.0;  /*initialize sum of partitions*/
    for(j = 0;j < NODO; j++)
      {
      temp_double = (double) occurrences[j];
      /*printf("%d ",occurrences[j]); DEBUG*/
      temp_double *= loga_n(temp_double);  /* Xj log Xj */
      double_sum += temp_double;  /* Sum of terms */
      }  /*end for*/
    /*printf("\n"); DEBUG*/
    double_sum = (double) double_sum / obj_cnt; /*Div by X */
    entropy[i] -= double_sum;
    entropy[i] /= loga_n(2.0);  /*convert result to log base 2 */
    printf("%11s     %15.12f\n", obj[i].name, entropy[i]);
    fprintf(outputfp,"%15.12f %3d\n", entropy[i], i + 1);
    }  /*end for*/
  /*Analyze system matrix to compute system entropy*/
  for(j = 0;j < obj_cnt; j++)
    rem_obj[j] = 0;  /*mark rows as not compared yet*/
  for(j = 0,NODO = 0;(j < (obj_cnt - 1)); j++)
    {  /*jth row is compared with mth row*/
    m = j + 1;
    if(rem_obj[j] != 1)  /*row not previously compared*/
      {
      rem_obj[j] = 1;  /*remove object from submatrix*/
      occurrences[NODO++] = 1;  /*initial occurrence*/
      while(m < obj_cnt)
        {  /*m is the following row index*/
        if(rem_obj[m] != 1)  /*row not matched before*/
          {
```

```
                for(k = 0;((OAT[j][k] == OAT[m][k]) &&
                  (k < asmp_cnt)); k++);  /*compare system rows*/
                if(k == asmp_cnt)  /*matching rows*/
                  {
                  occurrences[NODO - 1]++;  /*increase count*/
                  rem_obj[m] = 1;  /*remove object from submatrix*/
                  }  /*end if*/
                }  /*end if*/
              m++;  /*get ready for next row*/
              }  /*end while*/
            }  /*end if*/
          }  /*end for*/
        if((j == (obj_cnt - 1)) && (rem_obj[j] == 0))
          occurrences[NODO++] = 1;  /*last row is unique*/
        /*System Entropy Computation*/
        entropy[OBJECTS] = (double) obj_cnt;
        entropy[OBJECTS] = loga_n(entropy[OBJECTS]);  /*log |X|*/
        double_sum = 0.0;  /*initialize sum of partitions*/
        for(j = 0;j < NODO; j++)
          {
          temp_double = (double) occurrences[j];
          temp_double *= loga_n(temp_double);  /* Xj log Xj */
          double_sum += temp_double;  /* Sum of terms */
          }  /*end for*/
        double_sum = (double) double_sum / obj_cnt;  /*Div by X */
        entropy[OBJECTS] -= double_sum;
        entropy[OBJECTS] /= loga_n(2.0);  /*convert to log base 2*/
        for(i = 0, entropy_loading = 0.0;i < subsys_cnt; i++)
          entropy_loading += entropy[i];  /*compute entropy loading*/
        temp_double = (double) entropy_loading / subsys_cnt;
        printf("\nObject Count is %d\n", i);
        printf("Average Object Entropy is %15.12f\n", temp_double);
        printf("System Entropy (H(S)) is %15.12f\n",entropy[OBJECTS]);
        fprintf(outputfp, "%15.12f %3d\n", entropy[OBJECTS], i + 1);
        entropy_loading -= entropy[OBJECTS];  /*subtr. system entropy*/
        printf("System Entropy Loading (C(S)) is %15.12f\n",
          entropy_loading);
        temp_double = (double) entropy_loading / obj_cnt;
        printf("C(S) to Object Count Ratio is %15.12f\n",temp_double);
        fclose(datafp);  /*close input data file*/
        fclose(outputfp);  /*close output file*/
        }  /*end else*/
      }  /*end if*/
    }  /*end while*/
  }  /*end main*/


/*********************************************************************
*                                                                    *
*   This function accepts a double floating point number as input.   *
*   The output consists of the natural logarithm of the number       *
*   passed as input.  The output is of double floating point type.   *
*                                                                    *
*********************************************************************/
double loga_n(param)
```

```
  double param;
  {
  double temp[3];
  int i, j, k;

  temp[1] = 0.0;
  temp[2] = (param - 1.0) / (param + 1.0);
  for(i = 1;i < 160;i += 2)
    temp[1] += (double) power(temp[2], i) / i;
  temp[1] *= 2.0;
  return temp[1];
  }  /*end loga_n*/


/*********************************************************************
*                                                                    *
*   This function computes the nth power of a double floating point *
*   number.  The power can only be an integer.  The output is a     *
*   double floating point number.                                    *
*                                                                    *
*********************************************************************/
double power(base, exp)
  double base;
  int exp;
  {
  double temp;
  int i;

  for(i = 1, temp = 1.0;i <= exp; i++)
    temp *= base;
  return temp;
  }  /*end power*/
```

```
/*****************************************************************
*       L i n e s   o f   C o d e   M e t r i c   (C)           *
*****************************************************************
*                                                               *
*          File:  loc_c2.c                                      *
*          Author:  William R. Torres                           *
*          Date:  89/09/23                                      *
*          Class:  COMSC 5000 - Thesis                          *
*          Adviser:  Dr. Mansur Samadzadeh                      *
*                                                               *
*****************************************************************
*                                                               *
*   This program computes the lines of code for correct C       *
*   language source code.  The input for this program consists  *
*   of the filename of the file for which the lines of code     *
*   metric is desired.  The output is broken down into six      *
*   categories: actual C language lines of code, declaration    *
*   lines of code, non-declaration lines of code, lines of code *
*   with braces ('{' or '}'), comments lines, and blank lines.  *
*                                                               *
*   The following criteria is used to categorize the lines:     *
*   (1) all lines in which C statements are present, even if they *
*   include comments or multiple statements separated by        *
*   terminators (;), are counted as one line of code, (2) all   *
*   lines with comments and no C statements, even if the lines  *
*   have multiple comments, are counted as one comment line,    *
*   (3) all other lines are considered blank lines.             *
*                                                               *
*   The following criteria is used to subdivide the lines of    *
*   code: (1) all lines with a brace and no C statement are     *
*   counted as brace lines, (2) all lines in which a variable is *
*   declared are counted as declaration lines, and (3) all other *
*   lines are considered non-declaration lines.                 *
*                                                               *
*****************************************************************/

#include <stdio.h>
#include <ctype.h>
#define BUF_SIZE 80
#define TRUE 1
#define FALSE 0

main()
  {
  int loc,                      /*lines of code*/
      blank_cnt,                /*blank line count*/
      comm_cnt,                 /*comment line count*/
      comm_pres,                /*comment present*/
      instr_pres,               /*instruction present*/
      comm_counted,             /*line counted as comment*/
      brace_cnt,                /*brace line count*/
      brace_pres,               /*brace present*/
      decla_cnt,                /*declaration line count*/
      decla_pres,               /*declaration line presence*/
      decla_counted,            /*line counted as declaration*/
```

```
    i, j;
char inp_buffer[BUF_SIZE],  /*buffer for program lines*/
     file_buffer[25],       /*buffer for input program filename*/
     c;
FILE *filefp;               /*pointer to input program*/

strcpy(file_buffer, "");
while(strcmp(file_buffer, "quit") != 0)  /*main loop*/
  {
  printf("Enter filename or quit to stop program\n==>> ");
  scanf("%s", file_buffer);  /*get input filename*/
  if(strcmp(file_buffer, "quit") != 0)
    {
    if((filefp = fopen(file_buffer, "r")) == NULL)  /*open file*/
      printf("Error opening file %s\n", file_buffer);
    else  /*file opened for reading*/
      {
      loc = 0;  /*initialize lines of code count*/
      blank_cnt = 0;  /*initialize blank line count*/
      comm_cnt = 0;  /*initialize comment line count*/
      brace_cnt = 0;  /*initialize brace line count*/
      decla_cnt = 0;  /*initialize declaration line count*/
      comm_pres = FALSE;  /*no comment*/
      decla_pres = FALSE;  /*no declaration*/
      while(fgets(inp_buffer, BUF_SIZE, filefp))  /*read pgm. line*/
        {
        instr_pres = FALSE;  /*clear instruction presence*/
        comm_counted = FALSE;  /*line not counted as comment*/
        decla_counted = FALSE;  /*line not counted as declaration*/
        brace_pres = FALSE;  /*clear brace presence*/
        for(i = 0;inp_buffer[i] != '\n'; i++)
          {
          if((inp_buffer[i] == '/') && (inp_buffer[i + 1] == '*'))
            comm_pres = TRUE;  /*beginning of comment found*/
          c = inp_buffer[i];
          if(((c == 'c') || (c == 'd') || (c == 'f') || (c == 'i') ||
            (c == 'l') || (c == 's') || (c == 'F')) && (!comm_pres))
            {  /*1st letter in declarations*/
            if(isdecla(inp_buffer, i))
              decla_pres = TRUE;
            }  /*end if*/
          if((inp_buffer[i] == ';') && (decla_pres))
            {
            decla_pres = FALSE;  /*cancel decla. presence*/
            decla_counted = TRUE;  /*mark line as counted*/
            }  /*end if*/
          if((isgraph(inp_buffer[i])) && (!comm_pres) &&
            (inp_buffer[i] != '{') && (inp_buffer[i] != '}'))
            instr_pres = TRUE; /*print char, not sp, comm, or brace*/
          if((inp_buffer[i] == '{') || (inp_buffer[i] == '}') &&
            (!instr_pres) && (!comm_pres))
            brace_pres = TRUE;  /*brace found*/
          if((inp_buffer[i - 1] == '*') && (inp_buffer[i] == '/'))
            {
            comm_pres = FALSE;  /*cancel comment presence*/
```

```
            if((!instr_pres) && (!comm_counted) && (!brace_pres) &&
              (!decla_pres))
              {
              comm_cnt++;  /*add to comment count*/
              comm_counted = TRUE; /*mark line as counted for comm*/
              }  /*end if*/
            }  /*end if*/
          }  /*end for*/
        if((decla_pres) || (decla_counted))
          decla_cnt++;  /*add to declaration line to count*/
        else if(instr_pres)
          loc++;  /*add to lines of code*/
        else if((comm_pres) && (!brace_pres))
          comm_cnt++;  /*add comment line to count*/
        else if(brace_pres)
          brace_cnt++;  /*add to brace line count*/
        else if(!comm_counted)
          blank_cnt++;  /*add to blank line count*/
        if((comm_counted) && (instr_pres))
          comm_cnt--;  /*correct comm. count if intr. is found*/
        }  /*end while*/
      printf("\nTotal lines of code = %d\n", loc + decla_cnt +
brace_cnt);
      printf("    Total declaration lines of code = %d\n", decla_cnt);
      printf("    Total non-declaration lines of code = %d\n", loc);
      printf("    Total brace lines = %d\n", brace_cnt);
      printf("Total comment lines = %d\n", comm_cnt);
      printf("Total blank lines = %d\n", blank_cnt);
      printf("\nTotal program lines = %d\n\n",loc + decla_cnt +
        comm_cnt + brace_cnt + blank_cnt);
      fclose(filefp);  /*close input file*/
      }  /*end else*/
    }  /*end if*/
  }  /*end while*/
}  /*end main*/


/*****************************************************************
*                                                                *
*   This function is used to determine if a declaration line has *
*   been found.  This function is called whenever a character is *
*   found that matches the first character of the different      *
*   variable declaration types (e.g., char, int, float, etc.).   *
*   This program assumes that these keywords are always followed *
*   by at least one space in the input program.  Function        *
*   declaration lines are not counted as declaration lines.      *
*                                                                *
*   If a match is found, a TRUE condition is returned to the main *
*   program.  A FALSE condition is returned otherwise.           *
*                                                                *
*****************************************************************/
isdecla(buffer, i)
  char buffer[];
  int i;
  {
```

```
int j, type;
char c;

c = buffer[i];
type = FALSE;
if(c == 'c')  /*char*/
  {
  if((buffer[i + 1] == 'h') && (buffer[i + 2] == 'a') &&
    (buffer[i + 3] == 'r') && (buffer[i + 4] == ' '))
    {
    type = TRUE;
    j = i + 5;
    }  /*end if*/
  }  /*end if*/
else if(c == 'i')  /*int*/
  {
  if((buffer[i + 1] == 'n') && (buffer[i + 2] == 't') &&
    (buffer[i + 3] == ' '))
    {
    type = TRUE;
    j = i + 4;
    }  /*end if*/
  }  /*end else if*/
else if(c == 'd')  /*double*/
  {
  if((buffer[i + 1] == 'o') && (buffer[i + 2] == 'u') &&
    (buffer[i + 3] == 'b') && (buffer[i + 4] == 'l') &&
    (buffer[i + 5] == 'e') && (buffer[i + 6] == ' '))
    {
    type = TRUE;
    j = i + 7;
    }  /*end if*/
  }  /*end else if*/
else if(c == 'f')  /*float*/
  {
  if((buffer[i + 1] == 'l') && (buffer[i + 2] == 'o') &&
    (buffer[i + 3] == 'a') && (buffer[i + 4] == 't') &&
    (buffer[i + 5] == ' '))
    {
    type = TRUE;
    j = i + 6;
    }  /*end if*/
  }  /*end else if*/
else if(c == 'l')  /*long*/
  {
  if((buffer[i + 1] == 'o') && (buffer[i + 2] == 'n') &&
    (buffer[i + 3] == 'g') && (buffer[i + 4] == ' '))
    {
    type = TRUE;
    j = i + 5;
    }  /*end if*/
  }  /*end else if*/
else if(c == 's')  /*short*/
  {
  if((buffer[i + 1] == 'h') && (buffer[i + 2] == 'o') &&
```

```
    (buffer[i + 3] == 'r') && (buffer[i + 4] == 't') &&
    (buffer[i + 5] == ' '))
    {
    type = TRUE;
    j = i + 6;
    }  /*end if*/
  }  /*end else if*/
else if(c == 'F')  /*FILE*/
  {
  if((buffer[i + 1] == 'I') && (buffer[i + 2] == 'L') &&
    (buffer[i + 3] == 'E') && (buffer[i + 4] == ' '))
    {
    type = TRUE;
    j = i + 5;
    }  /*end if*/
  }  /*end else if*/
if(type == TRUE)
  {
  for(;buffer[j] != '\n';j++)
    if(buffer[j] == '(')  /*function declaration*/
      return (FALSE);
  return (TRUE);
  }  /*end if*/
return (FALSE);
}  /*end isdecla*/
```

```
/*****************************************************************
*      L i n e s   o f   C o d e   M e t r i c   (Ada)          *
*****************************************************************
*                                                               *
*          File:  loc_ada1.c                                    *
*          Author:  William R. Torres                           *
*          Date:  90/03/12                                      *
*          Class:  COMSC 5000 - Thesis                          *
*          Adviser:  Dr. Mansur Samadzadeh                      *
*                                                               *
*****************************************************************
*                                                               *
*   This program computes the lines of code for correct Ada     *
*   language source code.  The input for this program consists  *
*   of the filename of the file for which the lines of code     *
*   metric is desired.  The output is broken down into five     *
*   categories: actual Ada language lines of code, variable     *
*   declaration lines of code, non-declaration lines of code,   *
*   comments lines, and blank lines.                            *
*                                                               *
*   The following criteria is used to categorize the lines:     *
*   (1) all lines in which Ada statements are present, even if  *
*   they include comments or multiple statements separated by   *
*   terminators (;), are counted as one line of code, (2) all   *
*   lines with comments and no Ada statements are counted as one*
*   comment line, (3) all other lines are considered blank lines.*
*                                                               *
*   The following criteria is used to subdivide the lines of    *
*   code: (1) all lines in which a variable is declared are     *
*   counted as declaration lines and (2) all other lines are    *
*   counted as non-declaration lines.                           *
*                                                               *
*****************************************************************/

#include <stdio.h>
#include <ctype.h>
#define BUF_SIZE 80
#define TRUE 1
#define FALSE 0

main()
  {
  int loc,                      /*lines of code*/
      blank_cnt,                /*blank line count*/
      comm_cnt,                 /*comment line count*/
      comm_pres,                /*comment present*/
      instr_pres,               /*instruction present*/
      paren_count,              /*parenthesis count*/
      decla_cnt,                /*declaration line count*/
      decla_pres,               /*declaration line presence*/
      term_pres,                /*command terminator (;) presence*/
      i;
  char inp_buffer[BUF_SIZE],    /*buffer for program lines*/
       file_buffer[25];         /*buffer for input program filename*/
  FILE *filefp;                 /*pointer to input program*/
```

```
strcpy(file_buffer, "");
while(strcmp(file_buffer, "quit") != 0)  /*main loop*/
  {
  printf("Enter filename or quit to stop program\n==>> ");
  scanf("%s", file_buffer);  /*get input filename*/
  if(strcmp(file_buffer, "quit") != 0)
    {
    if((filefp = fopen(file_buffer, "r")) == NULL)  /*open file*/
      printf("Error opening file %s\n", file_buffer);
    else  /*file opened for reading*/
      {
      loc = 0;  /*initialize lines of code count*/
      blank_cnt = 0;  /*initialize blank line count*/
      comm_cnt = 0;  /*initialize comment line count*/
      decla_cnt = 0;  /*initialize declaration line count*/
      paren_count = 0;  /*clear parenthesis count*/
      while(fgets(inp_buffer, BUF_SIZE, filefp))  /*read pgm. line*/
        {
        instr_pres = FALSE;  /*clear instruction presence*/
        decla_pres = FALSE;  /*clear declaration presence*/
        comm_pres = FALSE;  /*no comment*/
        term_pres = FALSE;  /*clear terminator presence*/
        for(i = 0;inp_buffer[i] != '\n'; i++)
          {
          if((inp_buffer[i] == '-') && (inp_buffer[i + 1] == '-'))
            comm_pres = TRUE;  /*comment found*/
          if(inp_buffer[i] == '(')
            paren_count++;  /*opening parenthesis found*/
          if(inp_buffer[i] == ')')
            paren_count--;  /*closing parenthesis found*/
          if((inp_buffer[i] == ':') && (inp_buffer[i + 1] != '=') &&
            (paren_count == 0) && (!comm_pres))
            decla_pres = TRUE;  /*declaration found*/
          if(inp_buffer[i] == ';')
            term_pres = TRUE;  /*command terminator present*/
          if((isgraph(inp_buffer[i])) && (!comm_pres))
            instr_pres = TRUE; /*print char, not sp, or comm*/
          }  /*end for*/
        if((decla_pres) && (term_pres))
          decla_cnt++;  /*add to declaration line to count*/
        else if(instr_pres)
          loc++;  /*add to lines of code*/
        else if(comm_pres)
          comm_cnt++;  /*add comment line to count*/
        else
          blank_cnt++;  /*add to blank line count*/
        }  /*end while*/
      printf("\nTotal lines of code = %d\n", loc + decla_cnt);
      printf("    Total declaration lines = %d\n", decla_cnt);
      printf("    Total non-declaration lines = %d\n", loc);
      printf("Total comment lines = %d\n", comm_cnt);
      printf("Total blank lines = %d\n", blank_cnt);
      printf("\nTotal program lines = %d\n\n",loc + decla_cnt +
        comm_cnt + blank_cnt);
```

```
        fclose(filefp);  /*close input file*/
        }  /*end else*/
      }  /*end if*/
    }  /*end while*/
  }  /*end main*/
```

APPENDIX B

ADA PROGRAMS INCLUDED IN THE STUDY

```
--*****************************************************************
--*                  I n t e g e r   L i s t          (1 of 2)    *
--*****************************************************************
--*                                                               *
--*      File:  intmain.ada                                       *
--*      Author:  Ken Shumate                                     *
--*               "Understanding Ada with Abstract Data Types"    *
--*               John Wiley and Sons, 2nd ed., 1989              *
--*                                                               *
--*****************************************************************

    with TEXT_IO; use TEXT_IO;
    with Integer_List; use Integer_List;
    procedure Exercise_20_1 is
      package Int_IO is new INTEGER_IO(INTEGER);
      use Int_IO;

      Number : INTEGER;
    begin
      Initialize_List;
      PUT_LINE("Enter list of numbers terminated by -1");
      Create_List : loop
A.1     GET(Number);
A.2     exit Create_List when Number = -1;
A.3     Insert_At_Head(Number);
A.4     Insert_At_Tail(Number);
      end loop Create_List;

      PUT_LINE("The list of numbers is");
A.5   for I in 1..List_Length loop
A.6     PUT(Value_At_Position(I));
      end loop;
      NEW_LINE;

      PUT_LINE("Chopping the ends off");
      Delete_Head;
      Delete_Tail;

      PUT_LINE("The list of numbers is");
A.7   for I in 1..List_Length loop
A.8     PUT(Value_At_Position(I));
      end loop;
      NEW_LINE;
    end Exercise_20_1;
--*****************************************************************
--*                  I n t e g e r   L i s t          (2 of 2)    *
--*****************************************************************
--*                                                               *
--*      File:  intlist.ada                                       *
--*      Author:  Ken Shumate                                     *
--*               "Understanding Ada with Abstract Data Types"    *
--*               John Wiley and Sons, 2nd ed., 1989              *
--*                                                               *
--*****************************************************************
```

```
      package Integer_List is
        procedure Initialize_List;
        procedure Insert_At_Head(Value : in INTEGER);
        procedure Insert_At_Tail(Value : in INTEGER);
        procedure Delete_Head;
        procedure Delete_Tail;
        function Value_At_Position(Pos : in POSITIVE); return INTEGER;
        function List_Length return NATURAL;
      end Integer_List;

      package body Integer_List is
        type List;
        type Link is access List;
        type List is record
          Value : INTEGER;
          Next : Link;
        end record;

        Free, Head, Tail : Link;

        procedure Reclaim(P : in Link) is
        --add the node indicated by P onto the free list
        begin
B.1       if Free = null then
B.2         Free := P;
B.3         Free.Next := null;
B.4       else
B.5         P.Next := Free;
B.6         Free := P;
          end if;
        end Reclaim;

        function Alloc(Initial_Value : List) return Link is
        --allocate storage and initialize it
C.1       P : Link := Free;
        begin
C.2       if P = null then
C.3         P := new List;
C.4       else
C.5         Free := Free.Next;
          end if;
C.6       P.all := Initial_Value;
C.7       return P;
        end Alloc;

        procedure Initialize_List is
          P : Link;
        begin
D.1       while Head /= null loop
D.2         P := Head;
D3          Head := Head.Next;
D.4         Reclaim(P);
          end loop;
D.5       Tail := null;
        end Initialize_List;
```

```
      procedure Insert_At_Head(Value : in INTEGER) is
      begin
E.1     Head := Alloc(List'(Value, Head));
E.2     if Tail = null then  -- new list
E.3       Tail := Head;
        end if;
      end Insert_At_Head;

      procedure Insert_At_Tail(Value : in INTEGER) is
      begin
F.1     if Head = null then  -- first item to put into list
F.2       Head := Alloc(List'(Value, null));
F.3       Tail := Head;
F.4     else
F.5       Tail.Next := Alloc(List'(Value, null));  --tack onto end
F.6       Tail := Tail.Next;  --move tail to the new end of the list
        end if;
      end Insert_At_Tail;

      procedure Delete_Head is
G.1     P : Link := Head;
      begin
G.2     Head := Head.Next;
G.3     Reclaim(P);
G.4     if Head = null then  --deleted last item in list
G.5       Tail := null;
        end if;
      end Delete_Head;

      procedure Delete_Tail is
H.1     P : Link := Head;
      begin
H.2     if Head = Tail then --single item list
H.3       Head := null;
H.4       Tail := null;
H.5     else  --more than one item in list
H.6       while P.Next /= Tail loop
H.7         P := P.Next;
          end loop;
          --P now points to the next to last item in the list
H.8       Reclaim(Tail);
H.9       Tail := P;
H.10      Tail.Next := null;
        end if;
      end Delete_Tail;

      function Value_At_Position(Pos : in POSITIVE) return INTEGER is

I.1     P : Link := Head;
      begin
I.2     for I in 2..Pos loop
I.3       P := P.Next;
        end loop;
I.4     return P.Value;
```

```
      end Value_At_Position;

      function List_Length return NATURAL is
        Len : NATURAL := 0;
J.1     P : Link := Head;
      begin
J.2     while P /= null loop
J.3       Len := Len + 1;
J.4       P := P.Next;
        end loop;
J.5     return Len;
      end List_Length;
    end Integer_List;
```

```
--*************************************************************
--*   F o u r   F u n c t i o n   C a l c u l a t o r  (1 of 3)  *
--*************************************************************
--*                                                           *
--*      File:  calcmain.ada                                  *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern     *
--*               "Applied Ada"                               *
--*               Tab Professional and Reference Books, 1986  *
--*                                                           *
--*************************************************************

    with TEXT_IO, FPSTACK, FLOAT_CONV;
    procedure CALCULATE2 is
      use TEXT_IO, FPSTACK, FLOAT_CONV;
      package F_IO is new FLOAT_IO(FLOAT);
      subtype LINE40 is STRING(1..40);
      STR : LINE40 := (1..40 => ' ');
      NUM_VAL : FLOAT := 0.0;
      FIRST : BOOLEAN := TRUE;
      LEN : NATURAL;
      INVALID_ENTRY : exception;

      procedure DIRECTIONS is  --To display directions for use
      begin
        NEW_LINE;
        PUT_LINE("This is a simple calculator program.  It can ");
        PUT_LINE("add(+), subtract(-), multiply(*), and ");
        PUT_LINE("divide(/).  The calculator uses reverse Polish");
        PUT_LINE("notation and has a stack holding up to ten");
        PUT_LINE("floating point numbers.  Numbers and operators");
        PUT_LINE("must be entered one per line.  Enter 'R' for");
        PUT_LINE("reset to start over, '?' to get directions,");
        PUT_LINE("'D' to delete last entry, and 'Q' to quit.");
        NEW_LINE;
      end DIRECTIONS;

      procedure OPERATE(STRG : LINE40) is
        X, Y : FLOAT;
      begin
        case STRG(1) is
B.1       when '+' => POP(X); POP(Y); Y := X + Y;
B.2         PUSH(Y); F_IO.PUT(Y);
B.3       when '-' => POP(X); POP(Y); Y := Y - X;
B.4         PUSH(Y); F_IO.PUT(Y);
B.5       when '*' => POP(X); POP(Y); Y := X * Y;
B.6         PUSH(Y); F_IO.PUT(Y);
B.7       when '/' => POP(X); POP(Y); Y := Y / X;
B.8         PUSH(Y); F_IO.PUT(Y);
B.9       when 'r' | 'R' => CLEAR;
B.10      when 'd' | 'D' => POP(X);
B.11      when '?' => DIRECTIONS;
B.12      when 'q' | 'Q' => null;
B.13      when others => raise INVALID_ENTRY;
        end case;
        NEW_LINE;
```

```
      end OPERATE;

    begin  --Body of CALCULATE2
      DIRECTIONS;
A.1   while STR(1) /= 'Q' and STR(1) /= 'q' loop
        ERROR_SCOPE:  begin  --Block containing exception handler
A.2     if not FIRST then
          SKIP_LINE;
        end if;
A.3     GET_STRING(STR, LEN);
        NEW_LINE;
A.4     FIRST := FALSE;
A.5     if (STR(1) in '0' .. '9') or (STR(1) = '.') or
          (STR(1) = '-' and LEN > 1) then
A.6       STR_TO_FLT(STR, LEN, NUM_VAL);
A.7       PUSH(NUM_VAL);
A.8     else
A.9       OPERATE(STR);
        end if;
        exception  --Handler for block ERROR_SCOPE
A.10      when INVALID_ENTRY => PUT_LINE(" Invalid entry.");
A.11      when NUMERIC_ERROR =>
            PUT_LINE(" Attempt to divide by zero.");
        end ERROR_SCOPE;
      end loop;
    end CALCULATE2;
--*********************************************************************
--*   F o u r    F u n c t i o n    C a l c u l a t o r  (2 of 3)   *
--*********************************************************************
--*                                                                   *
--*      File:  calcstac.ada                                          *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern             *
--*               "Applied Ada"                                       *
--*               Tab Professional and Reference Books, 1986          *
--*                                                                   *
--*********************************************************************

    package FPSTACK is
      procedure CLEAR;  --Resets the stack
      procedure PUSH(NUM : FLOAT);
      procedure POP(NUM : out FLOAT);
      STACK_OVERFLOW, STACK_UNDERFLOW: exception;
    end FPSTACK;

    package body FPSTACK is
      NUM : FLOAT;
      LIMIT : constant NATURAL := 10;
      STACK : array(1 .. LIMIT) of FLOAT;
      TOP : NATURAL := 0;

      procedure PUSH(NUM : FLOAT) is
      begin
C.1     if TOP = LIMIT then
          raise STACK_OVERFLOW;
C.2     else
```

```
C.3         TOP := TOP + 1;
C.4         STACK(TOP) := NUM;
          end if;
        end PUSH;

        procedure POP(NUM : out FLOAT) is
        begin
D.1       if TOP = 0 then
            raise STACK_UNDERFLOW;
D.2       else
D.3         NUM := STACK(TOP);
D.4         TOP := TOP - 1;
          end if;
        end POP;

        procedure CLEAR is
        begin
E.1       TOP := 0;
        end CLEAR;
      end FPSTACK;
--*************************************************************************
--*   F o u r   F u n c t i o n   C a l c u l a t o r  (3 of 3)   *
--*************************************************************************
--*                                                                       *
--*      File:  calcfloa.ada                                              *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern                 *
--*               "Applied Ada"                                           *
--*               Tab Professional and Reference Books, 1986              *
--*                                                                       *
--*************************************************************************

    with TEXT_IO; use TEXT_IO;
    package FLOAT_CONV is
      subtype LINE40 is STRING(1 .. 40);
      INVALID_ENTRY : exception;
      procedure GET_STRING(STR : out LINE40;
                           LEN : out NATURAL);
      procedure STR_TO_FLT(STR : LINE40;
                           LEN : NATURAL;
                           NUM_VAL : out FLOAT);
    end FLOAT_CONV;

    package body FLOAT_CONV is
      procedure GET_STRING(STR : out LINE40;
                           LEN : out NATURAL) is
        CH : CHARACTER;
        CUM_COUNT : NATURAL := 0;
      begin
F.1     while not END_OF_LINE loop
F.2       GET(CH);
F.3       CUM_COUNT := CUM_COUNT + 1;
F.4       STR(CUM_COUNT) := CH;
        end loop;
F.5     LEN := CUM_COUNT;
      end GET_STRING;
```

```
      procedure STR_TO_FLT(STR : LINE40;
                           LEN : NATURAL;
                           NUM_VAL : out FLOAT) IS
        X : FLOAT := 0.0;
        SIGN : FLOAT := 1.0;
        DECIMAL_POINT : BOOLEAN := FALSE;
        COUNT : INTEGER := 0;
        EXP : INTEGER := 0;
        EXP_SIGN : INTEGER := 1;
        INDEX : INTEGER := 1;
        CH, CHR : CHARACTER;

      begin
G.1     if STR(1) = '-' then
G.2       SIGN := -1.0;
G.3       INDEX := 2;
        end if;
G.4     while INDEX <= LEN loop
G.5       CH := STR(INDEX);
          case CH is
G.6         when '.' => DECIMAL_POINT := TRUE;
G.7         when '0'..'9' => X := X * 10.0
              + FLOAT(CHARACTER'POS(CH) - CHARACTER'POS('0'));
G.8           if DECIMAL_POINT then
G.9             COUNT := COUNT + 1;
              end if;
G.10        when 'E' | 'e' =>
G.11          for JDEX in (INDEX + 1)..LEN loop
G.12            CHR := STR(JDEX);
                case CHR is
G.13              when '0'..'9' =>
                    EXP := EXP * 10 + CHARACTER'POS(CHR)
                      - CHARACTER'POS('0');
G.14              when '-' => EXP_SIGN := -1;
G.15              when others => raise INVALID_ENTRY;
                end case;
              end loop;
G.16          INDEX := LEN;
G.17        when others => raise INVALID_ENTRY;
          end case;
G.18      INDEX := INDEX + 1;
        end loop;
G.19    NUM_VAL := SIGN * X * 10.0**(EXP_SIGN * EXP - COUNT);
      end STR_TO_FLT;
    end FLOAT_CONV;
```

```
--*****************************************************************
--*                  A d d r e s s    B o o k          (1 of 12)   *
--*****************************************************************
--*                                                               *
--*      File:  addrdec.ada                                       *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern         *
--*               "Applied Ada"                                   *
--*               Tab Professional and Reference Books, 1986      *
--*                                                               *
--*****************************************************************

    with DIRECT_IO;
    with TEXT_IO; use TEXT_IO;

    package ADDRDEC is
      subtype LINE40 is STRING(1..40);
      type ADDRESS is record
        NAME : STRING(1..40) := (1..40 => ' ');
        STREET : STRING(1..40) := (1..40 => ' ');
        CITY : STRING(1..20) := (1..20 => ' ');
        STATE : STRING(1..2) := "  ";
        ZIP : STRING(1..5) := (1..5 => ' ');
        AREA : STRING(1..3) := "   ";
        PHONE : STRING(1..8) := (1..8 => ' ');
      end record;
      type KEY is record
        NAME : STRING(1..40) := (1..40 => ' ');
        PT_DATA : POSITIVE;
      end record;
      MAX_SIZE : constant := 20;  --Maximum size of deletion array
      type INT_ARRAY is array (POSITIVE range <>) of INTEGER;
      type KEY_ARRAY is array (POSITIVE range <>) of KEY;
      type LIST(SIZE : NATURAL) is record
        LAST_REC : INTEGER;
        NEXT_SPACE : INTEGER;
        SPACE : INT_ARRAY(1..MAX_SIZE);
        KEY_LIST : KEY_ARRAY(1..SIZE);
      end record;
      type A_LIST is access LIST;

      package ADDRESS_IO is new DIRECT_IO(ADDRESS);
      package INDEX_IO is new DIRECT_IO(LIST);

      type OPERATION is (CREATE, ADD, DELETE, CHANGE, SEARCH, QUIT);
      QUITTING : exception;
      INDX_ID : INDEX_IO.FILE_TYPE;
      DATA_ID : ADDRESS_IO.FILE_TYPE;
      DATA_NAME : constant STRING := "ADDRBK1";
      INDX_NAME : constant STRING := "ADDRINDX1";
    end ADDRDEC;
--*****************************************************************
--*                  A d d r e s s    B o o k          (2 of 12)   *
--*****************************************************************
--*                                                               *
--*      File:  addrmain.ada                                      *
```

```
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern          *
--*               "Applied Ada"                                    *
--*               Tab Professional and Reference Books, 1986       *
--*                                                                *
--******************************************************************

    with TEXT_IO; use TEXT_IO;
    with GET_STRING;
    with DISPLAY;
    with ADDRDEC; use ADDRDEC;
    with START_UP;
    with ENTER_DATA;
    with CREATE_LIST;
    with SELECT_ALTERNATIVE;
    with SEARCH;
    with ALTER_DATA;
    with INSERT;
    with DELETE;

    procedure ADDRESS_BOOK is
    NAME : STRING(1..40);
    PT_LIST : A_LIST;
    LEN_LIST : A_LIST;
    DATA : ADDRESS;
    INDEX : INTEGER;
    FOUND : BOOLEAN;
    OP : OPERATION;
    FIRST : BOOLEAN := TRUE;

    procedure GET_NAME(NAME : in out STRING) is
    COUNT : INTEGER;
    begin
      PUT_LINE("Enter name(last, first)");
      SKIP_LINE;
B.1   GET_STRING(NAME, COUNT);
B.2   for I in COUNT + 1..NAME'LAST loop
B.3     NAME(I) := ' ';
      end loop;
      return;
    end GET_NAME;

    begin  --Open Files and Load Index from File
A.1   PT_LIST := new LIST(0);
A.2   LEN_LIST := new LIST(0);
A.3   START_UP(PT_LIST, LEN_LIST, FIRST);
      loop
A.4     SELECT_ALTERNATIVE(OP, FIRST);
        case OP is
A.5       when CREATE => CREATE_LIST(PT_LIST, LEN_LIST);
A.6       when ADD => ENTER_DATA(DATA);
A.7                   INSERT(DATA, PT_LIST);
A.8       when CHANGE => GET_NAME(NAME);
A.9                      SEARCH(SEEK_NAME => NAME,
                         PT_LIST => PT_LIST,
                         DATA => DATA,
```

```
                          INDEX => INDEX,
                          FOUND => FOUND);
A.10                      if FOUND then
A.11                        ALTER_DATA(DATA);
A.12                        ADDRESS_IO.WRITE(DATA_ID, DATA,
                              ADDRESS_IO.POSITIVE_COUNT
                              (PT_LIST.KEY_LIST(INDEX).PT_DATA));
A.13                      else
                            PUT_LINE("Name not found.");
                          end if;
A.14        when DELETE => GET_NAME(NAME);
A.15                      DELETE(NAME, PT_LIST);
A.16        when SEARCH => GET_NAME(NAME);
A.17                      SEARCH(SEEK_NAME => NAME,
                          PT_LIST => PT_LIST,
                          DATA => DATA,
                          INDEX => INDEX,
                          FOUND => FOUND);
A.18                      if FOUND then
A.19                        DISPLAY(DATA);
A.20                      else
                            PUT_LINE("Name not found.");
                          end if;
A.21        when QUIT => null;
          end case;
A.22      exit when OP = QUIT;
        end loop;
A.23    LEN_LIST.LAST_REC := PT_LIST.SIZE;
A.24    INDEX_IO.WRITE(INDX_ID, LEN_LIST.all, 1);
A.25    INDEX_IO.WRITE(INDX_ID, PT_LIST.all, 2);
A.26    INDEX_IO.CLOSE(INDX_ID);
A.27    ADDRESS_IO.CLOSE(DATA_ID);

    end ADDRESS_BOOK;
--*************************************************************************
--*                    A d d r e s s    B o o k          (3 of 12)   *
--*************************************************************************
--*                                                                       *
--*      File:  startup.ada                                               *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern                 *
--*               "Applied Ada"                                           *
--*               Tab Professional and Reference Books, 1986              *
--*                                                                       *
--*************************************************************************

    with TEXT_IO; use TEXT_IO;
    with ADDRDEC; use ADDRDEC;
    with GET_STRING;
    with CREATE_LIST;
    procedure START_UP(PT_LIST : in out A_LIST;
      LEN_LIST : in out A_LIST; FIRST : in out BOOLEAN) is
    RESPONSE : STRING(1..40);
    LEN : NATURAL;

    begin
```

```
    OPEN_INDX :
    begin
C.1   INDEX_IO.OPEN(INDX_ID, INDEX_IO.INOUT_FILE, INDX_NAME);
C.2   INDEX_IO.READ(INDX_ID, LEN_LIST.all, 1);
C.3   PT_LIST := new LIST(LEN_LIST.LAST_REC);
C.4   INDEX_IO.READ(INDX_ID, PT_LIST.all, 2);
      exception
C.5     when INDEX_IO.NAME_ERROR =>
          PUT_LINE("There is no Index File. Do you");
          PUT_LINE("want to create one (C) or quit (Q)?");
C.6       GET_STRING(RESPONSE, LEN);
C.7       FIRST := FALSE;
C.8       if RESPONSE(1) = 'Q' or RESPONSE(1) = 'q' then
            raise QUITTING;
C.9       else
C.10        CREATE_LIST(PT_LIST, LEN_LIST);
          end if;
    end OPEN_INDX;

    OPEN_DATA :
    begin
C.11  if not ADDRESS_IO.IS_OPEN(DATA_ID) then
C.12    ADDRESS_IO.OPEN(DATA_ID, ADDRESS_IO.INOUT_FILE, DATA_NAME);
      end if;
      exception
C.13    when ADDRESS_IO.NAME_ERROR =>
          PUT_LINE("There is no Data File. Do you");
          PUT_LINE("want to create one (C) or quit (Q)?");
C.14      if FIRST then
C.15        FIRST := FALSE;
C.16      else
            SKIP_LINE;
          end if;
C.17      GET_STRING(RESPONSE, LEN);
C.18      if RESPONSE(1) = 'Q' or RESPONSE(1) = 'q' then
            raise QUITTING;
C.19      else
C.20        INDEX_IO.DELETE(INDX_ID);
C.21        CREATE_LIST(PT_LIST, LEN_LIST);
          end if;
    end OPEN_DATA;

    end START_UP;
--*************************************************************************
--*                    A d d r e s s    B o o k          (4 of 12)   *
--*************************************************************************
--*                                                                       *
--*      File:  crelist.ada                                               *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern                 *
--*               "Applied Ada"                                           *
--*               Tab Professional and Reference Books, 1986              *
--*                                                                       *
--*************************************************************************

    with TEXT_IO; use TEXT_IO;
```

```
    with ADDRDEC; use ADDRDEC;
    with ENTER_DATA;
    procedure CREATE_LIST(PT_LIST : in out A_LIST;
      LEN_LIST : in out A_LIST) is
    RESPONSE : STRING(1..40);
    INIT_KEY : KEY;
    DATA : ADDRESS;

    begin
    FILE1_GEN :
    begin
D.1   ADDRESS_IO.CREATE(DATA_ID, ADDRESS_IO.INOUT_FILE, DATA_NAME);
      exception
D.2     when ADDRESS_IO.STATUS_ERROR =>
D.3       ADDRESS_IO.DELETE(DATA_ID);
D.4       ADDRESS_IO.CREATE(DATA_ID, ADDRESS_IO.INOUT_FILE, DATA_NAME);
    end FILE1_GEN;

    FILE2_GEN :
    begin
D.5   INDEX_IO.CREATE(INDX_ID, INDEX_IO.INOUT_FILE, INDX_NAME);
      exception
D.6     when INDEX_IO.STATUS_ERROR =>
D.7       INDEX_IO.DELETE(INDX_ID);
D.8       INDEX_IO.CREATE(INDX_ID, INDEX_IO.INOUT_FILE, INDX_NAME);
    end FILE2_GEN;

D.9   ENTER_DATA(DATA);
D.10  ADDRESS_IO.WRITE(DATA_ID, DATA, 1);
D.11  INIT_KEY.NAME := DATA.NAME;
D.12  INIT_KEY.PT_DATA := 1;
D.13  PT_LIST := new LIST'(SIZE => 1,
                           LAST_REC => 1,
                           NEXT_SPACE => 0,
                           SPACE => (1..MAX_SIZE => 0),
                           KEY_LIST => KEY_ARRAY'(1 => INIT_KEY));
D.14  LEN_LIST.LAST_REC := 1;
D.15  LEN_LIST.NEXT_SPACE := 0;
D.16  LEN_LIST.SPACE := (LEN_LIST.SPACE'RANGE => 0);

    end CREATE_LIST;
--*********************************************************************
--*                  A d d r e s s   B o o k          (5 of 12)   *
--*********************************************************************
--*                                                                   *
--*      File:  entdata.ada                                           *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern             *
--*               "Applied Ada"                                       *
--*               Tab Professional and Reference Books, 1986          *
--*                                                                   *
--*********************************************************************

    with TEXT_IO; use TEXT_IO;
    with ADDRDEC; use ADDRDEC;
    with GET_STRING;
```

```
     with ALTER_DATA;
     with DISPLAY;

     procedure ENTER_DATA(DATA : out ADDRESS) is
     RESPONSE : STRING(1..40);
     COUNT : INTEGER;
     NEW_ADDRESS : ADDRESS;

     begin
       PUT_LINE("Enter name of addressee(last, first).");
       SKIP_LINE;
E.1    GET_STRING(NEW_ADDRESS.NAME, COUNT);

       PUT_LINE("Street address");
       SKIP_LINE;
E.2    GET_STRING(NEW_ADDRESS.STREET, COUNT);

       PUT_LINE("City");
       SKIP_LINE;
E.3    GET_STRING(NEW_ADDRESS.CITY, COUNT);

       PUT_LINE("Two-letter abbreviation for state");
       SKIP_LINE;
E.4    GET_STRING(NEW_ADDRESS.STATE, COUNT);

       PUT_LINE("Five digit zip code");
       SKIP_LINE;
E.5    GET_STRING(NEW_ADDRESS.ZIP, COUNT);

       PUT_LINE("Phone area code");
       SKIP_LINE;
E.6    GET_STRING(NEW_ADDRESS.AREA, COUNT);

       PUT_LINE("Phone number");
       SKIP_LINE;
E.7    GET_STRING(NEW_ADDRESS.PHONE, COUNT);

E.8    DISPLAY(NEW_ADDRESS);
       PUT_LINE("Is this correct? (Y/N)");
       SKIP_LINE;
E.9    GET_STRING(RESPONSE, COUNT);
E.10   if RESPONSE(1) /= 'Y' and RESPONSE(1) /= 'y'
E.11     then ALTER_DATA(NEW_ADDRESS);
       end if;

E.12   DATA := NEW_ADDRESS;

     end ENTER_DATA;
--*************************************************************
--*                  A d d r e s s   B o o k          (6 of 12)  *
--*************************************************************
--*                                                            *
--*      File:  display.ada                                    *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern      *
--*               "Applied Ada"                                *
```

```
--*              Tab Professional and Reference Books, 1986        *
--*                                                                *
--******************************************************************

    with TEXT_IO; use TEXT_IO;
    with ADDRDEC; use ADDRDEC;

    procedure DISPLAY(DATA : ADDRESS) is
    begin
F.1   PUT_LINE(DATA.NAME);
F.2   PUT_LINE(DATA.STREET);
F.3   PUT(DATA.CITY);PUT(", ");PUT(DATA.STATE);
F.4   PUT(" ");PUT(DATA.ZIP);NEW_LINE;
F.5   PUT(DATA.AREA);PUT("-");PUT(DATA.PHONE);NEW_LINE;

    end DISPLAY;
--******************************************************************
--*                  A d d r e s s    B o o k          (7 of 12)   *
--******************************************************************
--*                                                                *
--*      File:  altdata.ada                                        *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern          *
--*               "Applied Ada"                                    *
--*               Tab Professional and Reference Books, 1986       *
--*                                                                *
--******************************************************************

    with TEXT_IO; use TEXT_IO;
    with ADDRDEC; use ADDRDEC;
    with GET_STRING;
    with DISPLAY;

    procedure ALTER_DATA(DATA : in out ADDRESS) is
    NUM_CHAR : NATURAL;
    RESPONSE : STRING(1..40);

    procedure ALTER_FIELD(STRG : in out STRING) is
    REPLY : STRING(STRG'range) :=
            (STRG'FIRST..STRG'LAST => ' ');
    IN_CHAR : NATURAL;
    begin
H.1   PUT_LINE(STRG);
      SKIP_LINE;
H.2   GET_STRING(REPLY, IN_CHAR);
H.3   if IN_CHAR > 0 then
H.4     STRG := REPLY;
      end if;
    end ALTER_FIELD;

    begin  -- ALTER_DATA
      loop
        PUT_LINE("For each line push carriage return without");
        PUT_LINE("entry to leave line unchanged.  Otherwise,");
        PUT_LINE("enter a new line.");
        NEW_LINE;
```

```
G.1     ALTER_FIELD(DATA.NAME);
G.2     ALTER_FIELD(DATA.STREET);
G.3     ALTER_FIELD(DATA.CITY);
G.4     ALTER_FIELD(DATA.STATE);
G.5     ALTER_FIELD(DATA.ZIP);
G.6     ALTER_FIELD(DATA.AREA);
G.7     ALTER_FIELD(DATA.PHONE);
        PUT_LINE("Address is now: ");
        NEW_LINE;
G.8     DISPLAY(DATA);
        NEW_LINE;
        PUT_LINE("Is this correct? (Y/N)");
G.9     RESPONSE(1) := 'Y';
        SKIP_LINE;
G.10    GET_STRING(RESPONSE, NUM_CHAR);
G.11    if RESPONSE(1) = 'Y' or RESPONSE(1) = 'y' then
          return;
        end if;
      end loop;

    end ALTER_DATA;
--*************************************************************************
--*                  A d d r e s s    B o o k          (8 of 12)   *
--*************************************************************************
--*                                                                       *
--*      File:  selalte.ada                                               *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern                 *
--*               "Applied Ada"                                           *
--*               Tab Professional and Reference Books, 1986              *
--*                                                                       *
--*************************************************************************

    with TEXT_IO, ADDRDEC;
    use TEXT_IO, ADDRDEC;
    with GET_STRING;

    procedure SELECT_ALTERNATIVE(MODE : out OPERATION;
                                 FIRST : in out BOOLEAN) is
    RESPONSE : STRING(1..40);
    COUNT : INTEGER;

    begin
      loop
        PUT_LINE("Data base operations are:");
        NEW_LINE;
        PUT_LINE("  INITIALIZE          DELETE");
        PUT_LINE("  CHANGE              SEARCH");
        PUT_LINE("  ADD                 QUIT");
        NEW_LINE;
        PUT_LINE("Enter first character of selection.");
I.1     if not FIRST then
          SKIP_LINE;
        end if;

I.2     GET_STRING(RESPONSE, COUNT);
```

```
I.3     FIRST := FALSE;
        case RESPONSE(1) is
I.4       when 'I' | 'i' => MODE := CREATE;
            PUT_LINE("Initialize replaces your address files!");
            PUT_LINE("Do you want to continue? (Y/N)");
            SKIP_LINE;
I.5         GET_STRING(RESPONSE, COUNT);
I.6         if RESPONSE(1) = 'Y' or RESPONSE(1) = 'y' then
              return;
            end if;
I.7       when 'C' | 'c' => MODE := CHANGE; return;
I.8       when 'A' | 'a' => MODE := ADD; return;
I.9       when 'D' | 'd' => MODE := DELETE; return;
I.10      when 'S' | 's' => MODE := SEARCH; return;
I.11      when 'Q' | 'q' => MODE := QUIT;
            PUT_LINE("QUIT (Y/N)?");
            SKIP_LINE;
I.12        GET_STRING(RESPONSE, COUNT);
I.13        if RESPONSE(1) = 'Y' or RESPONSE(1) = 'y' then
              return;
            end if;
I.14      when others => null;
        end case;
      end loop;

    end SELECT_ALTERNATIVE;
--*************************************************************
--*                  A d d r e s s   B o o k          (9 of 12)  *
--*************************************************************
--*                                                           *
--*      File:  insert.ada                                    *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern     *
--*               "Applied Ada"                               *
--*               Tab Professional and Reference Books, 1986  *
--*                                                           *
--*************************************************************

    with TEXT_IO; use TEXT_IO;
    with ADDRDEC; use ADDRDEC;
    with DISPLAY;
    with GET_STRING;
    with SEARCH;

    procedure INSERT(DATA : ADDRESS; PT_LIST : in out A_LIST) is
    TEMP_DATA : ADDRESS;
    INDEX, COUNT, REC_NUM : NATURAL;
    FOUND : BOOLEAN;
    RESPONSE : STRING(1..40);
    NEW_LIST : A_LIST;

    begin
J.1   SEARCH(DATA.NAME, PT_LIST, TEMP_DATA, INDEX, FOUND);
J.2   if FOUND then
        PUT_LINE("There is an address for this name.  It is: ");
        NEW_LINE;
```

```
J.3      DISPLAY(TEMP_DATA);
         PUT_LINE("Overwrite (O) or Leave (L) this address?");
         SKIP_LINE;
J.4      GET_STRING(RESPONSE, COUNT);
J.5      if RESPONSE(1) /= 'O' AND RESPONSE(1) /= 'o' then
           return;  --terminate insertion
J.6      else --overwrite the existing record
J.7        REC_NUM := PT_LIST.KEY_LIST(INDEX).PT_DATA;
         end if;
J.8    else -- no address found
         -- allocate a new index list with space for a new entry
J.9      NEW_LIST := new LIST(PT_LIST.SIZE + 1);
J.10     for I in 1..INDEX loop
           -- copy entries up to one preceding insertion
J.11       NEW_LIST.KEY_LIST(I) := PT_LIST.KEY_LIST(I);
         end loop;
J.12     INDEX := INDEX + 1;
J.13     NEW_LIST.KEY_LIST(INDEX).NAME := DATA.NAME;
         -- Where should data be written?
J.14     if PT_LIST.NEXT_SPACE = 0 then -- append to file
J.15       NEW_LIST.LAST_REC := PT_LIST.LAST_REC + 1;
J.16       REC_NUM := NEW_LIST.LAST_REC;
J.17       NEW_LIST.NEXT_SPACE := 0;
J.18     else -- a deleted address can be overwritten
J.19       NEW_LIST.LAST_REC := PT_LIST.LAST_REC;
J.20       REC_NUM := PT_LIST.SPACE(PT_LIST.NEXT_SPACE);
J.21       NEW_LIST.NEXT_SPACE := PT_LIST.NEXT_SPACE - 1;
         end if;
J.22     NEW_LIST.SPACE := PT_LIST.SPACE;
J.23     NEW_LIST.KEY_LIST(INDEX).PT_DATA :=REC_NUM;
         --copy entries following insertion
J.24     for I in INDEX+1..NEW_LIST.SIZE loop
J.25       NEW_LIST.KEY_LIST(I) := PT_LIST.KEY_LIST(I - 1);
         end loop;
J.26     PT_LIST := NEW_LIST; -- access new list with PT_LIST
       end if;
J.27   ADDRESS_IO.WRITE(DATA_ID, DATA,
                        ADDRESS_IO.POSITIVE_COUNT(REC_NUM));

    end INSERT;
--*************************************************************************
--*                    A d d r e s s    B o o k          (10 of 12)    *
--*************************************************************************
--*                                                                       *
--*      File:  search.ada                                                *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern                 *
--*               "Applied Ada"                                           *
--*               Tab Professional and Reference Books, 1986              *
--*                                                                       *
--*************************************************************************

    with TEXT_IO; use TEXT_IO;
    with ADDRDEC; use ADDRDEC;

    procedure SEARCH(SEEK_NAME : STRING; PT_LIST : A_LIST;
```

```
        DATA : out ADDRESS; INDEX : out NATURAL;
        FOUND : out BOOLEAN) is
      THIS_NAME : STRING(SEEK_NAME'RANGE);
      LAST : INTEGER := 0;
      LLAST : INTEGER := 0;
      HIGH : INTEGER := PT_LIST.SIZE;
      LOW : INTEGER := 1;
      NEXT : INTEGER := (HIGH + LOW)/2;
      BLANKS : ADDRESS;

      begin
K.1     while (NEXT /= LAST) and (NEXT /= LLAST) loop
K.2       THIS_NAME := PT_LIST.KEY_LIST(NEXT).NAME;
K.3       if SEEK_NAME = THIS_NAME then
K.4         FOUND := TRUE;
K.5         INDEX := NEXT;
K.6         ADDRESS_IO.READ(DATA_ID, DATA, ADDRESS_IO.
              POSITIVE_COUNT(PT_LIST.KEY_LIST(NEXT).PT_DATA));
            return;
K.7       elsif SEEK_NAME > THIS_NAME then
K.8         LOW := NEXT;
K.9         LLAST := LAST;
K.10        LAST := NEXT;
K.11        NEXT := (NEXT + HIGH + 1)/2;
K.12      else -- SEEK_NAME < THIS_NAME
K.13        HIGH := NEXT;
K.14        LLAST := LAST;
K.15        LAST := NEXT;
K.16        NEXT := (NEXT + LOW)/2;
          end if;
        end loop;
K.17    FOUND := FALSE;
K.18    DATA := BLANKS;
K.19    if SEEK_NAME > THIS_NAME then
K.20      INDEX := LAST;
K.21    else  -- SEEK_NAME < THIS_NAME
K.22      INDEX := LAST - 1;
        end if;
      end SEARCH;
--*********************************************************************
--*                  A d d r e s s   B o o k          (11 of 12)      *
--*********************************************************************
--*                                                                   *
--*      File:  delete.ada                                            *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern             *
--*               "Applied Ada"                                       *
--*               Tab Professional and Reference Books, 1986          *
--*                                                                   *
--*********************************************************************

    with TEXT_IO; use TEXT_IO;
    with DISPLAY;
    with ADDRDEC; use ADDRDEC;
    with GET_STRING;
    with SEARCH;
```

```
    procedure DELETE(DEL_NAME : STRING;
                     PT_LIST : in out A_LIST) is
    RESPONSE : STRING(1..40) := (1..40 => ' ');
    TEMP_DATA : ADDRESS;
    INDEX, COUNT, REC_NUM : INTEGER;
    FOUND : BOOLEAN;
    NEW_LIST : A_LIST;

    begin
L.1   SEARCH(SEEK_NAME => DEL_NAME,
             PT_LIST => PT_LIST,
                DATA => TEMP_DATA,
               INDEX => INDEX,
               FOUND => FOUND);
L.2   if not FOUND then
        PUT_LINE("Name not found.");
        return;
L.3   else  --delete index entry and address record
        NEW_LINE;
L.4     DISPLAY(TEMP_DATA);
        PUT_LINE("Do you want to delete this address? (Y/N");
        SKIP_LINE;
L.5     GET_STRING(RESPONSE, COUNT);
L.6     if RESPONSE(1) /= 'Y' and RESPONSE(1) /= 'y' then
          return;
        end if;

        --delete entry from index
L.7     REC_NUM := PT_LIST.KEY_LIST(INDEX).PT_DATA;
L.8     for I in INDEX..PT_LIST.SIZE - 1 loop
L.9       PT_LIST.KEY_LIST(I) := PT_LIST.KEY_LIST(I + 1);
        end loop;

        --make a new list that is one entry shorter
L.10    NEW_LIST := new LIST'(PT_LIST.SIZE - 1, PT_LIST.LAST_REC,
          PT_LIST.NEXT_SPACE, PT_LIST.SPACE,
          PT_LIST.KEY_LIST(1..PT_LIST.SIZE - 1));

        --add record number to stack of space available
L.11    if NEW_LIST.NEXT_SPACE < NEW_LIST.SPACE'LAST then
L.12      NEW_LIST.NEXT_SPACE := NEW_LIST.NEXT_SPACE + 1;
L.13      NEW_LIST.SPACE(NEW_LIST.NEXT_SPACE) := REC_NUM;
        end if;
L.14    PT_LIST := NEW_LIST; --access new list with PT_LIST
      end if;

    end DELETE;
--*************************************************************
--*                  A d d r e s s   B o o k        (12 of 12)  *
--*************************************************************
--*                                                           *
--*      File:  getstr.ada                                    *
--*      Author:  Gerald L. Mohnkern and Beverly Mohnkern     *
--*               "Applied Ada"                               *
```

```
--*              Tab Professional and Reference Books, 1986          *
--*                                                                  *
--********************************************************************

    with TEXT_IO; use TEXT_IO;
    with ADDRDEC; use ADDRDEC;
    procedure GET_STRING(STR : out LINE40;
                         LEN : out NATURAL) is
      CH : CHARACTER;
      CUM_COUNT : NATURAL := 0;
    begin
M.1   while not END_OF_LINE loop
M.2     GET(CH);
M.3     CUM_COUNT := CUM_COUNT + 1;
M.4     STR(CUM_COUNT) := CH;
      end loop;
M.5   LEN := CUM_COUNT;
    end GET_STRING;
```

APPENDIX C

C PROGRAMS INCLUDED IN THE STUDY

```
/*****************************************************************
*                        F a s t f i n d                        *
*****************************************************************
*                                                               *
*      File:  fastfind.c                                        *
*      Author:  Webb Miller                                     *
*               "A Software Tools Sampler"                      *
*               Prentice-Hall, Inc., 1987                       *
*                                                               *
*****************************************************************
*                                                               *
*  fastfind - print lines containing a given pattern string.    *
*                                                               *
*  Program description:                                         *
*                                                               *
*    A command line has the form                                *
*      fastfind pat [file1] [file2] ....                        *
*    where pat is any sequence of characters.  A character pair *
*    "\n" at the beguinning or end of pat matches the beguinning*
*    or end of a text line.  If no file is named, then standard *
*    input is read.  If more than one file is named, then each  *
*    printed line is preceded by its file's name.               *
*                                                               *
*  Portability:                                                 *
*                                                               *
*    Files are read with the UNIX routines:                     *
*                                                               *
*      int read(fd, buffer, max_chars)                          *
*        int fd, max_chars;                                     *
*        char buffer[];                                         *
*          Read at most max_chars characters from the file with *
*          file descriptor fd to the buffer.  The returned value*
*          is the actual number of characters read.             *
*                                                               *
*  File lines should be separated by '\n'.                      *
*                                                               *
*****************************************************************/

    #include <stdio.h>
    #define MAX_NAME 50

    /*Two buffers of length BUFSIZ (defined in stdio.h are arranged as
      a circle; the location just left of the current buffer is the end
      of the other buffer.
    */

    #define TO_LEFT         buf[1 - curbuf] + BUFSIZ - 1
    #define DECREMENT(x)    if(x == buf[curbuf]) x = TO_LEFT; else --x
    #define INCREMENT(x)    if(x == TO_LEFT) x = buf[curbuf]; else ++x
    #define PROG_NAME       "fastfind1"

    int  curbuf,            /*current buffer*/
         fd,                /*file descriptor*/
         nfile,             /*number of files*/
         shift[128];        /*shift table*/
```

```
        char buf[2][BUFSIZ],    /*text buffers*/
             *end_pat,          /*last position in the pattern*/
             *file,             /*name of the file*/
             *lim,              /*limit for search*/
             *pat,              /*pattern*/
             *pos;              /*search pointer to the text*/

        char prog_name[MAX_NAME + 1];  /*used in error messages*/

        main(argc, argv)
          int argc;
          char *argv[];
          {
          int i, length;
          char *p;

A.1       savename(PROG_NAME);  /*for error messages*/
A.2       if(argc == 1)
A.3         fatal("No pattern was given.");
A.4       pat = argv[1];
          /*handle new line characters in the pattern*/
A.5       if(pat[0] == '\\' && pat[1] == 'n')
A.6         *++pat = '\n';
A.7       if((length = strlen(pat)) == 0)
A.8         fatal("Pattern length is zero.");
A.9       if(length > 1 && pat[length - 2] == '\\' && pat[length - 1] == 'n')
            {
A.10        pat[length - 2] = '\n';
A.11        pat[length - 1] = '\0';
A.12        --length;
            }  /*end if*/
A.13      end_pat = pat + length - 1;
A.14      for(i = 0;i < 128; ++i)
A.15        shift[i] = length;
A.16      for(p = pat; *p != '\0'; ++p)
A.17        shift[*p & 0177] = --length;
A.18      if((nfile = argc - 2) == 0)
            {
A.19        fd = 0;  /*standard input*/
            scan();
            }  /*end if*/
A.20      else
            {
A.21        for(i = 2;i < argc; ++i)  /*for each specified file*/
              {
A.22          file = argv[i];
A.23          if((fd = open(file, 0)) < 0)
A.24            fprintf(stderr, "%s: Cannot open %s.\n", PROG_NAME, file);
A.25          else
                {
                scan();
A.26            close(fd);
                }  /*end else*/
              }  /*end for*/
```

```
        }  /*end else*/
      exit(0);
      }  /*end main*/


    /*scan - find lines in file that contains the pattern string*/
    scan()
      {
      int increment;

B.1   buf[1][BUFSIZ - 1] = '\n';  /*in case the first line matches*/
B.2   curbuf = 1;
B.3   lim = pos = buf[1];  /*force an immediate call to fill_buffer()*/
      for(;;)
        {
        /*Pos points to a text position that might end on an occurrence
of
          pat.  If that character differs from the last character of pat,
          then pos is shifted to the next position that might yield a
          match.  The shifting stops when pos reaches the end of the
buffer
          or an instance of the last pattern character.
        */
B.4     while(pos < lim && (increment = shift[*pos & 0177]) > 0)
B.5       pos += increment;
B.6     if(pos < lim)  /*shifting ended with pos in buffer*/
          {
B.7       if(is_match())
            print_line();
B.8       ++pos;
          }  /*end if*/
        /*else past end of buffer; fill the other buffer*/
B.9     else if(fill_buffer() == EOF)
          break;
        }  /*end for*/
      }  /*end scan*/


    /*fill_buffer - fill other buffer; points pos to first char read;
point
      lim just beyond the last character read; return EOF at the end of
      the file.
    */
    fill_buffer()
      {
C.1   curbuf = 1 - curbuf;
C.2   pos = buf[curbuf];
C.3   if((lim = pos + read(fd, pos, BUFSIZ)) == pos)
        return(EOF);
      return(!EOF);
      }  /*end fill_buffer*/


    /*is_match - tell if a copy of the pattern ends at pos*/
    is_match()
```

```
      {
D.1   char *t = pos, *p = end_pat;  /*already know that *t == *p */

D.2   while(--p >= pat)
        {
D.3     DECREMENT(t);
D.4     if(*p != *t)
          return(0);
        }  /*end while*/
D.5   return(1);
      }  /*end is_match*/


    /*print_line - print the line pointed to by pos; move pos to the end
      of line.
    */
    print_line()
      {
      char *t;

E.1   if(nfile > 1)
E.2     printf("%s:", file);
E.3   if(*pos == '\n')  /*find the start of the line*/
E.4     DECREMENT(pos);
E.5   for(t = pos; *t != '\n'; )
E.6     DECREMENT(t);
      /*print the portion of the line before the match*/
E.7   while(t != pos)
        {
E.8     INCREMENT(t);
E.9     putchar(*t);
        }  /*end while*/
      /*print the portion of the line after the match*/
E.10  while(*pos != '\n')
        {
E.11    if(++pos >= lim && fill_buffer() == EOF)
          break;
E.12    putchar(*pos);
        }  /*end while*/
      }  /*end print_line*/


    /*savename - record a program name for error messages*/
    savename(name)
      char *name;
      {
      char *strcpy();

F.1   if(strlen(name) <= MAX_NAME)
F.2     strcpy(prog_name, name);
      }  /*end savename*/


    /*fatal - print message and die*/
    fatal(msg)
```

```
        char *msg;
        {
G.1     if(prog_name[0] != '\0')
G.2       fprintf(stderr, "%s: ",prog_name);
G.3     fprintf(stderr, "%s\n",msg);
        exit(1);
        }  /*end fatal*/
```

```
/*********************************************************************
*                  M a i l i n g    L i s t                         *
*********************************************************************
*                                                                   *
*      File:  mail.c                                                *
*      Author:  Herbert Schildt                                     *
*               "Advanced Turbo C"                                  *
*               Osborne McGraw-Hill, 1987                           *
*                                                                   *
*********************************************************************/

    #include "stdio.h"

    struct address {
      char      .30];
      char street[40];
      char city[20];
      char state[3];
      char zip[10];  /*hold US & Canadian zips*/
      struct address *next;  /*pointer to next entry*/
      struct address *prior;  /*pointer to previous entry*/
      }  list_entry;

    struct address *start;  /*pointer to first entry in list*/
    struct address *last;  /*pointer to last entry*/

    void enter(), display(), search(), save(), load();

    main()
      {
      char s[80], choice;
      struct address *info;

A.1   start = last = NULL;  /*zero length list*/
      for(;;)
        {
        switch(menu_select())
          {
A.2       case 1: enter();
            break;
A.3       case 2: delete();
            break;
A.4       case 3: list();
            break;
A.5       case 4: search();  /*find a name*/
            break;
A.6       case 5: save();  /*save list to disk*/
            break;
A.7       case 6: load();  /*read from disk*/
            break;
A.8       case 7: exit(0);
          }  /*end switch*/
        }  /*end for*/
      }  /*end main*/
```

```
      /* select an operation */
      menu_select()
        {
        char s[80];
        int c;

        printf("1. Enter a name\n");
        printf("2. Delete a name\n");
        printf("3. List the file\n");
        printf("4. Search\n");
        printf("5. Save the file\n");
        printf("6. Load the file\n");
        printf("7. Quit\n");
        do
          {
          printf("\nEnter your choice: ");
B.1       gets(s);
B.2       c = atoi(s);
B.3       } while(c < 1 || c > 7);
        return c;
        } /*end menu_select*/


      /* enter name and address */
      void enter()
        {
        struct address *info, *dls_store();

        for(;;)
          {
C.1       info = (struct address *) malloc (sizeof(list_entry));
C.2       if(!info)
            {
            printf("\nOut of memory\n");
            return;
            } /*end if*/
C.3       inputs("  enter name: ",info->name,30);
C.4       if(!info->name[0]) break;  /*stop entering*/
C.5       inputs("enter street: ",info->street,40);
C.6       inputs("  enter city: ",info->city,20);
C.7       inputs(" enter state: ",info->state,3);
C.8       inputs("   enter zip: ",info->zip,10);
C.9       start = dls_store(info, start);
          } /*end for*/
        } /*end enter*/


      /* This function will input a string up to the length in count.
         This will prevent the string from overrunning its space and
         display a prompt message.
      */
      inputs(prompt, s, count)
        char *prompt;
        char *s;
```

```
      int count;
      {
      char p[255];

      do
        {
        printf(prompt);
D.1     gets(p);
D.2     if(strlen(p) > count)
          printf("\ntoo long\n");
D.3     } while(strlen(p) > count);
        strcpy(s,p);
      }  /*end inputs*/


    /* This function creates a doubly linked list in sorted order.  A
       pointer to the first element is returned because it is possible
       that a new element will be inserted at the start of the list.
    */
    struct address *dls_store(i,top)
      struct address *i;  /*new element*/
      struct address *top;  /*first element in list*/
      {
      struct address *old, *p;

E.1   if(last == NULL)  /*first element in list*/
        {
E.2     i->next = NULL;
E.3     i->prior = NULL;
E.4     last = i;
E.5     return i;
        }  /*end if*/
E.6   p = top;  /*start at top of list*/
E.7   old = NULL;
E.8   while(p)
        {
E.9     if(strcmp(p->name, i->name) < 0)
          {
E.10      old = p;
E.11      p = p->next;
          }  /*end if*/
E.12    else
          {
E.13      if(p->prior)
            {
E.14          p->prior->next = i;
E.15          i->next = p;
E.16          i->prior = p->prior;
E.17          p->prior = i;
E.18          return top;
            }  /*end if*/
E.19      i->next = p;  /*new first element*/
E.20      i->prior = NULL;
E.21      p->prior = i;
E.22      return i;
```

```
          }  /*end else*/
        }  /*end while*/
E.23  old->next = i;  /*put on end*/
E.24  i->next = NULL;
E.25  i->prior = old;
E.26  last = i;
E.27  return start;
      }  /*end dls_store*/


    /* Remove an element from the list. */
    delete()
      {
      struct address *info, *find();
      char s[80];

      printf("enter name: ");
F.1   gets(s);
F.2   info = find(s);
F.3   if(info)
        {
F.4     if(start == info)
          {
F.5       start = info->next;
F.6       if(start)
F.7         start->prior = NULL;
F.8       else
F.9         last = NULL;
          }  /*end if*/
F.10    else
          {
F.11      info->prior->next = info->next;
F.12      if(info != last)
F.13        info->next->prior = info->prior;
F.14      else
F.15        last = info->prior;
          }  /*end else*/
F.16    free(info);  /*return memory space to system*/
        }  /*end if*/
      }  /*end delete*/


    struct address *find(name)
      char *name;
      {
      struct address *info;

G.1   info = start;
G.2   while(info)
        {
G.3     if(!strcmp(name, info->name))
G.4       return info;
G.5     info = info->next;  /*get next address*/
        }  /*end while*/
      printf("name not found\n");
```

```
      return NULL;  /*not found*/
      }  /*end find*/


    list()
      {
      register int t;
      struct address *info;

H.1   info = start;
H.2   while(info)
        {
H.3     display(info);
H.4     info = info->next;  /*get next address*/
        }  /*end while*/
      printf("\n\n");
      }  /*end list*/


    void display(info)
      struct address *info;
      {
I.1   printf("%s\n",info->name);
I.2   printf("%s\n",info->street);
I.3   printf("%s\n",info->city);
I.4   printf("%s\n",info->state);
I.5   printf("%s\n",info->zip);
      printf("\n\n");
      }  /*end display*/


    void search()
      {
      char name[40];
      struct address *info, *find();

      printf("enter name to find: ");
J.1   gets(name);
J.2   if(!(info = find(name)))
        printf("not found\n");
J.3   else
J.4     display(info);
      }  /*end search*/


    void save()
      {
      register int t;
      struct address *info;
      FILE *fp;

K.1   if((fp = fopen("mail_list","wb")) == NULL)
        {
        printf("Cannot open file\n");
        exit(1);
```

```
        } /*end if*/
      printf("\nSaving file\n");
K.2   info = start;
K.3   while(info)
        {
K.4     fwrite(info,sizeof(struct address),1,fp);
K.5     info = info->next;  /*get next address*/
        } /*end while*/
K.6   fclose(fp);
      } /*end save*/


    void load()
      {
      register int t;
      struct address *info, *temp = NULL;
      FILE *fp;

L.1   if((fp = fopen("mail_list","rb")) == NULL)
        {
        printf("Cannot open file\n");
        exit(1);
        } /*end if*/
L.2   while(start)
        {
L.3     info = start->next;
L.4     free(info);
L.5     start = info;
        } /*end while*/
      printf("\nLoading file\n");
L.6   start = (struct address *) malloc (sizeof(struct address));
L.7   if(!start)
        {
        printf("Out of memory\n");
        return;
        } /*end if*/
L.8   info = start;
L.9   while(!feof(fp))
        {
L.10    if(1 != fread(info,sizeof(struct address),1,fp))
          break;
        /*get memory for next*/
L.11    info->next = (struct address *) malloc(sizeof(struct address));
L.12    if(!info->next)
          {
          printf("Out of memory\n");
          return;
          } /*end if*/
L.13    info->prior = temp;
L.14    temp = info;
L.15    info = info->next;
        } /*end while*/
L.16  temp->next = NULL;  /*last entry*/
L.17  last = temp;
L.18  start->prior = NULL;
```

```
L.19  fclose(fp);
      }  /*end load*/
```

```
/*********************************************************************
*                    T e x t    E d i t o r                          *
*********************************************************************
*                                                                   *
*      File:  editor.c                                              *
*      Author:  Herbert Schildt                                     *
*               "Advanced Turbo C"                                  *
*               Osborne McGraw-Hill, 1987                           *
*                                                                   *
*********************************************************************/

    #include "stdio.h"
    #include "ctype.h"

    struct line
      {
      char text[81];
      int num;  /*line number of line*/
      struct line *next;  /*pointer to next entry*/
      struct line *prior;  /*pointer to previous entry*/
      };
    struct line *start;  /*pointer to first entry in list*/
    struct line *last;  /*pointer to last entry*/
    struct line *dls_store(), *find();
    void patchup(), delete(), list(), save(), load();

    main(argc, argv)
      int argc;
      char *argv[];
      {
      char s[80], choice, fname[80];
      struct line *info;
      int linenum = 1;

A.1   start = NULL;  /*zero length list*/
A.2   last = NULL;
A.3   if(argc == 2)
A.4     load(argv[1]);  /*read file on command line*/
      do
        {
A.5     choice = menu_select();
        switch(choice)
          {
A.6       case 1: printf("Enter line number: ");
A.7         gets(s);
A.8         linenum = atoi(s);
A.9         linenum = enter(linenum);
            break;
A.10      case 2: delete();
            break;
A.11      case 3: list();
            break;
A.12      case 4: printf("Enter filename: ");
A.13        gets(fname);
A.14        save(fname);  /*write to disk*/
```

```
            break;
A.15       case 5: printf("Enter filename: ");
A.16          gets(fname);
A.17          load(fname);  /*read from disk*/
              break;
A.18       case 6: exit(0);
           }  /*end switch*/
         }  while(1);
       }  /*end main*/


     /* Select a menu option */
     menu_select()
       {
       char s[80];
       int c;

       printf("1. Enter text\n");
       printf("2. Delete a line\n");
       printf("3. List the file\n");
       printf("4. Save the file\n");
       printf("5. Load the file\n");
       printf("6. Quit\n");
       do
         {
         printf("\nEnter your choice: ");
B.1      gets(s);
B.2      c = atoi(s);
B.3      }  while(c < 1 || c > 6);
       return c;
       }  /*end menu_select*/


     /* Enter text at linenum */
     enter(linenum)
       int linenum;
       {
       struct line *info;
       char t[81];

       do  /*entry loop*/
         {
C.1      info = (struct line *) malloc (sizeof'struct line));
C.2      if(!info)
           {
           printf("\nOut of memory\n");
           return linenum;
           }  /*end if*/
C.3      printf("%d : ",linenum);
C.4      gets(info->text);
C.5      info->num = linenum;
C.6      if(*info->text)
           {
C.7        if(find(linenum))
C.8          patchup(linenum, 1);  /*fix up old line numbers*/
```

```
C.9         if(*info->text)
C.10          start = dls_store(info);
            }  /*end if*/
C.11      else
            break;
C.12      linenum++;
          }  while(1);
        return linenum;
        }  /*end enter*/


      /*This function increases line numbers by 1 of lines below an
        inserted line and decreases line numbers by 1 of lines after
        deleted lines.
      */
      void patchup(n, incr)
        int n;
        int incr;
        {
        struct line *i;

D.1     i = find(n);
D.2     while(i)
          {
D.3       i->num = i->num + incr;
D.4       i = i->next;
          }  /*end while*/
        }  /*end patchup*/


      /* Store in sorted order by line number */
      struct line *dls_store(i)
        struct line *i;
        {
        struct line *old, *p;

E.1     if(last == NULL)  /*first element in list*/
          {
E.2       i->next = NULL;
E.3       i->prior = NULL;
E.4       last = i;
E.5       return i;
          }  /*end if*/
E.6     p = start;  /*start at top of list*/
E.7     old = NULL;
E.8     while(p)
          {
E.9       if(p->num < i->num)
            {
E.10        old = p;
E.11        p = p->next;
            }  /*end if*/
E.12      else
            {
E.13        if(p->prior)
```

```
                {
E.14            p->prior->next = i;
E.15            i->next = p;
E.16            i->prior = p->prior;
E.17            p->prior = i;
E.18            return start;
                }  /*end if*/
E.19          i->next = p;  /*new first element*/
E.20          i->prior = NULL;
E.21          p->prior = i;
E.22          return i;
              }  /*end else*/
            }  /*end while*/
E.23      old->next = i;  /*put on end*/
E.24      i->next = NULL;
E.25      i->prior = old;
E.26      last = i;
E.27      return start;
          }  /*end dls_store*/


        /* Delete a line */
        void delete()
          {
          struct line *info;
          char s[80];
          int linenum;

          printf("Enter line number: ");
F.1       gets(s);
F.2       linenum = atoi(s);
F.3       info = find(linenum);
F.4       if(info)
            {
F.5         if(start == info)
              {
F.6           start = info->next;
F.7           if(start)
F.8             start->prior = NULL;
F.9           else
F.10            last = NULL;
              }  /*end if*/
F.11        else
              {
F.12          info->prior->next = info->next;
F.13          if(info != last)
F.14            info->next->prior = info->prior;
F.15          else
F.16            last = info->prior;
              }  /*end else*/
F.17        free(info);  /*return memory space to system*/
F.18        patchup(linenum + 1, -1);  /*decrement line numbers*/
            }  /*end if*/
          }  /*end delete*/
```

```
    /* Find a line of text */
    struct line *find(linenum)
      int linenum;
      {
      struct line *info;

G.1   info = start;
G.2   while(info)
        {
G.3     if(linenum == info->num)
G.4       return info;
G.5     info = info->next;  /*get next address*/
        }  /*end while*/
      return NULL;  /*not found*/
      }  /*end find*/


    /* List the text */
    void list()
      {
      struct line *info;

H.1   info = start;
H.2   while(info)
        {
H.3     printf("%d: %s\n",info->num, info->text);
H.4     info = info->next;  /*get next address*/
        }  /*end while*/
      printf("\n\n");
      }  /*end list*/


    /* Save the file */
    void save(fname)
      char *fname;
      {
      register int t;
      struct line *info;
      char *p;
      FILE *fp;

I.1   if((fp = fopen(fname, "w")) == NULL)
        {
        printf("Cannot open file\n");
        exit(0);
        }  /*end if*/
      printf("\nSaving file\n");
I.2   info = start;
I.3   while(info)
        {
I.4     p = info->text;  /*convert to char pointer*/
I.5     while(*p)
I.6       putc(*p++, fp);  /*save byte at a time*/
I.7     putc('\r', fp);  /*terminator*/
```

```
I.8       putc('\n', fp);  /*terminator*/
I.9       info = info->next;  /*get next line*/
          }  /*end while*/
I.10    fclose(fp);
        }  /*end save */



      /* Load the file */
      void load(fname)
        char *fname;
        {
        register int t, size, lnct;
        struct line *info, *temp;
        char *p;
        FILE *fp;

J.1     if((fp = fopen(fname, "r")) == NULL)
          {
          printf("Cannot open file\n");
          exit(0);
          }  /*end if*/
J.2     while(start)  /*free any previous edit*/
          {
J.3       temp = start;
J.4       start = start->next;
J.5       free(temp);
          }  /*end while*/
        printf("\nLoading file\n");
J.6     size = sizeof(struct line);
J.7     start = (struct line *) malloc (size);
J.8     if(!start)
          {
          printf("Out of memory\n");
          return;
          }  /*end if*/
J.9     info = start;
J.10    p = info->text;  /*convert to char pointer*/
J.11    lnct = 1;
J.12    while((*p = getc(fp)) != EOF)
          {
J.13      if(!isprint(*p))
            break;
J.14      p++;
J.15      while((*p = getc(fp)) != '\r')
J.16        p++;
J.17      getc(fp);  /*throw away the \n */
J.18      *p = '\0';
J.19      info->num = lnct++;
J.20       info->next = (struct line *) malloc (size); /*get memory for
next*/
J.21      if(!info->next)
            {
            printf("Out of memory\n");
            return;
```

```
          }  /*end if*/
J.22     info->prior = temp;
J.23     temp = info;
J.24     info = info->next;
J.25     p = info->text;
         }  /*end while*/
J.26   temp->next = NULL;  /*last entry*/
J.27   last = temp;
J.28   free(info);
J.29   start->prior = NULL;
J.30   fclose(fp);
       }  /*end load*/
```

APPENDIX D

ENTROPY LOADING DATA TABLES

TABLE IV

ASSUMPTIONS FOR THE C PROGRAM
FASTFIND, CASE 1

Asmp. Number	Assumption
1	There exists a function called fill_buffer().
2	There exists a function called is_match().
3	There exists a function called print_line().
4	There exists a function called savename() with one parameter.
5	The parameter for the function savename() has read access only.
6	There exists a function called fatal() with one parameter.
7	The parameter for the function fatal() has read access only.
8	Read access to character string pointer called end_pat that points to the last character in pattern.
9	Read access to integer table called shift.
10	Read access to curbuf, index to the text buffer in use.
11	Write access to curbuf, index to the text buffer in use.
12	Read access to buf, buffers used to store text read from the files being searched.
13	Write access to buf, buffers used to store text read from the files being searched.
14	Read access to lim, character pointer that points to the last character read from the file being searched.
15	Write access to lim, character pointer that points to the last character read from the file being searched.
16	Read access to pos, character pointer that points to the character being compared with the input pattern.
17	Write access to pos, character pointer that points to the character being compared with the input pattern.
18	Function is_match returns 1 if a match to the input pattern has been found in the file being searched, and returns 0 otherwise.
19	Function fill_buffer returns EOF when the an attempt is made to read from the file being searched beyond the end of the file.

TABLE IV (Continued)

Asmp. Number	Assumption
20	Read access to prog_name, character string that holds the name of the executable program, i.e. (fastfind1).
21	Write access to prog_name, character string that holds the name of the executable program, i.e. (fastfind1).
22	Write access to character string name.
23	Read access to integer variable argc.
24	Variable argc equals 1.
25	Write access to character string msg.
26	Write access to character string pointer pat that points to the pattern to be matched.
27	Read access to character string pointer argv[].
28	Read access to character string pointer pat that points to the pattern to be matched.
29	The first character in the string pointed at by pat is '\'.
30	The second character in the string pointed at by pat is 'n'.
31	Read access to character string pointed at by pat.
32	Write access to character string pointed at by pat.
33	Read access to integer variable length.
34	Write access to integer variable length.
35	Variable length equals 0.
36	Variable length is greater than 1.
37	The next to last character in the string pointed at by pat is '\'.
38	The last character in the string pointed at by pat is 'n'.
39	Write access to character string pointer called end_pat that points to the last character in pattern.
40	Read access to integer variable i.
41	Write access to integer variable i.
42	Variable i is less than 128.
43	Write access to integer array shift.
44	The character being read from the string pointed at by the pointer p is not the line terminator ('\0').
45	Write access to character string pointer p.
46	Read access to character string pointer p.
47	Read access to the character string pointed at by p.
48	Variable nfile equals 0.
49	Read access to integer nfiles holding the number of files to be searched for pattern.

TABLE IV (Continued)

Asmp. Number	Assumption
50	Write access to integer nfiles holding the number of files to be searched for pattern.
51	Read access to character string file holding the name of the file presently being searched.
52	Write access to character string file holding the name of the file presently being searched.
53	Read access to file pointer fd that points to the file presently open.
54	Write access to file pointer fd that points to the file presently open.
55	Complement of assumption number 48.
56	i < argc, i.e., there are more files to be searched.
57	File pointed at by fd can't be opened for reading.
58	Complement of assumption number 57.

TABLE V

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE C PROGRAM FASTFIND, CASE 1

Object	Assumptions
A.1	4, 5
A.2	23, 24
A.3	6, 7
A.4	26, 27
A.5	29, 30, 31
A.6	32
A.7	31, 33, 35, 36
A.8	5
A.9	31, 33, 36, 37, 38
A.10	31, 33
A.11	31, 33
A.12	34
A.13	28, 33, 39
A.14	40, 41, 42
A.15	33, 40, 43
A.16	28, 44, 45, 46, 47
A.17	34, 35, 43
A.18	23, 48, 49, 50
A.19	54
A.20	23, 49, 50, 55
A.21	40, 41, 56
A.22	27, 40, 52
A.23	51, 53, 54, 57
A.24	51
A.25	51, 53, 54, 58
A.26	53
B.0	1, 2, 3, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19
C.0	10, 11, 12, 15, 16, 17, 53
D.0	8, 10, 12, 16, 28
E.0	1, 10, 12, 14, 16, 17, 19, 49, 51
F.0	5, 21
G.0	7, 20

TABLE VI

ASSUMPTIONS FOR THE C PROGRAM
FASTFIND, CASE 2

Asmp. Number	Assumption
1	There exists a module called module_1 with one parameter.
2	The parameter in module module_1 has read access only.
3	There exists a function called is_match().
4	There exists a function called savename() with one parameter.
5	The parameter for the function savename() has read access only.
6	There exists a function called fatal() with one parameter.
7	The parameter for the function fatal() has read access only.
8	Read access to character string pointer called end_pat that points to the last character in pattern.
9	Read access to integer array called shift.
10	Read access to curbuf, index to the text buffer in use.
11	Write access to curbuf, index to the text buffer in use.
12	Read access to buf, buffers used to store text read from the files being searched.
13	Write access to buf, buffers used to store text read from the files being searched.
14	Read access to lim, character pointer that points to the last character read from the file being searched.
15	Write access to lim, character pointer that points to the last character read from the file being searched.
16	Read access to pos, character pointer that points to the character being compared with the input pattern.
17	Write access to pos, character pointer that points to the character being compared with the input pattern.
18	Function is_match returns 1 if a match to the input pattern has been found in the file being searched, and returns 0 otherwise.

TABLE VI (Continued)

Asmp. Number	Assumption
19	Module module_1 returns EOF when the an attempt is made to read from the file being searched beyond the end of the file.
20	Read access to prog_name, character string that holds the name of the executable program, i.e. (fastfind1).
21	Write access to prog_name, character string that holds the name of the executable program, i.e. (fastfind1).
22	Write access to character string name.
23	Read access to integer variable argc.
24	Variable argc equals 1.
25	Write access to character string msg.
26	Write access to character string pointer pat that points to the pattern to be matched.
27	Read access to character string pointer argv[].
28	Read access to character string pointer pat that points to the pattern to be matched.
29	The first character in the string pointed at by pat is '\'.
30	The second character in the string pointed at by pat is 'n'.
31	Read access to character string pointed at by pat.
32	Write access to character string pointed at by pat.
33	Read access to integer variable length.
34	Write access to integer variable length.
35	Variable length equals 0.
36	Variable length is greater than 1.
37	The next to last character in the string pointed at by pat is '\'.
38	The last character in the string pointed at by pat is 'n'.
39	Write access to character string pointer called end_pat that points to the last character in pattern.
40	Read access to integer variable i.
41	Write access to integer variable i.
42	Variable i is less than 128.
43	Write access to integer array shift.
44	The character being read from the string pointed at by the pointer p is not the line terminator ('\0').
45	Write access to character string pointer p.
46	Read access to character string pointer p.
47	Read access to the character string pointed at by p.
48	Variable nfile equals 0.

TABLE VI (Continued)

Asmp. Number	Assumption
49	Read access to integer nfiles holding the number of files to be searched for pattern.
50	Write access to integer nfiles holding the number of files to be searched for pattern.
51	Read access to character string file holding the name of the file presently being searched.
52	Write access to character string file holding the name of the file presently being searched.
53	Read access to file pointer fd that points to the file presently open.
54	Write access to file pointer fd that points to the file presently open.
55	Complement of assumption number 48.
56	i < argc, i.e., there are more files to be searched.
57	File pointed at by fd can't be opened for reading.
58	Complement of assumption number 57.
59	Read access to character string pointed at by pos.
60	Write access to integer variable increment.
61	Read access to integer variable increment.
62	Pointer pos is less than pointer lim, i.e., the character pointed at by pos is closer to the beginning of the file than the character pointed at by lim.
63	Variable increment is larger than zero.
64	Module module_1 is in fill_buffer mode.

TABLE VII

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE C PROGRAM FASTFIND, CASE 2

Object	Assumptions
A.1	4, 5
A.2	23, 24
A.3	6, 7
A.4	26, 27
A.5	29, 30, 31
A.6	32
A.7	31, 33, 35, 36
A.8	5
A.9	31, 33, 36, 37, 38
A.10	31, 33
A.11	31, 33
A.12	34
A.13	28, 33, 39
A.14	40, 41, 42
A.15	33, 40, 43
A.16	28, 44, 45, 46, 47
A.17	34, 35, 43
A.18	23, 48, 49, 50
A.19	54
A.20	23, 49, 50, 55
A.21	40, 41, 56
A.22	27, 40, 52
A.23	51, 53, 54, 57
A.24	51
A.25	51, 53, 54, 58
A.26	53
B.1	13
B.2	11
B.3	12, 15, 16, 17
B.4	9, 12, 14, 59, 60, 61, 62, 63
B.5	16, 17, 61
B.6	14, 16, 62
B.7	3, 18
B.8	16, 17
B.9	1, 2, 19, 64
CE.0	10, 11, 12, 14, 15, 16, 17, 49, 51, 53
D.0	8, 10, 12, 16, 28
F.0	5, 21
G.0	7, 20

TABLE VIII

ASSUMPTIONS FOR THE C PROGRAM
FASTFIND, CASE 3

Asmp. Number	Assumption
1	Write access to character string name.
2	Read access to character string name.
3	Write access to character string prog_name.
4	Read access to integer variable argc.
5	Variable argc equals 1.
6	Write access to character string msg.
7	Read access to character string prog_name.
8	First character in character string prog_name is not the line terminator ('\0').
9	Read access to character string msg.
10	Write access to character string pointer pat that points to the pattern to be matched.
11	Read access to character string pointer argv[].
12	Read access to character string pointer pat that points to the pattern to be matched.
13	The first character in the string pointed at by pat is '\'.
14	The second character in the string pointed at by pat is 'n'.
15	Read access to character string pointed at by pat.
16	Write access to character string pointed at by pat.
17	Read access to integer variable length.
18	Write access to integer variable length.
19	Variable length equals 0.
20	Variable length is greater than 1.
21	The next to last character in the string pointed at by pat is '\'.
22	The last character in the string pointed at by pat is 'n'.
23	Write access to character string pointer end_pat.
24	Read access to integer variable i.
25	Write access to integer variable i.
26	Variable i is less than 128.
27	Write access to integer array shift.
28	The character being read from the string pointed at by the pointer p is not the line terminator ('\0').
29	Write access to character string pointer p.
30	Read access to character string pointer p.
31	Read access to the character string pointed at by p.
32	Variable nfile equals 0.
33	Read access to integer variable nfile.
34	Write access to integer variable nfile.

TABLE VIII (Continued)

Asmp. Number	Assumption
35	Write access to integer file descriptor fd.
36	Write access to character string buf.
37	Write access to integer variable curbuf.
38	Read access to character string pointer pos.
39	Write access to character string pointer pos.
40	Write access to character string pointer lim.
41	Read access to character string buf.
42	Read access to integer array shift.
43	Read access to character string pointed at by pos.
44	Write access to integer variable increment.
45	Read access to integer variable increment.
46	pos < lim, i.e., the character pointed at by pos is closer to the beginning of the file than the character pointed at by lim.
47	Variable increment is greater than 0.
48	Read access to character string pointer lim.
49	Read access to character string pointer end_pat.
50	Write access to the character string pointed at by p.
51	Write access to the character string pointed at by t.
52	p >= pat, i.e., the character pointed at by p is at the same location as the character pointed at by pat or at a location after the location of the character pointed at by pat.
53	Read access to character string pointer t.
54	Write access to character string pointer t.
55	Read access to integer variable curbuf.
56	Variable t equals buf[curbuf].
57	Character being read from the string pointed at by p is not the same character being read from the string pointed at by t.
58	Complement of assumption number 52.
59	Character string pointed at by p is the same as the character string pointed at by t.
60	nfile > 1, i.e., more than one file is to be searched for the same pattern.
61	Read access to character string pointed at by pointer file.
62	The character being read from the string pointed at by pos is the newline character ('\n').
63	Variable pos equals buf[curbuf].
64	Complement of assumption number 62.
65	Read access to character string pointed at by t.

TABLE VIII (Continued)

Asmp. Number	Assumption
66	Pointers t and pos are not equal, i.e., point to different locations in the file.
67	t = buf[1 - curbuf] + BUFSIZ - 1.
68	The character being read from the string pointed at by pos is not the newline character ('\n').
69	Complement of assumption number 46.
70	The end of the file being searched has been reached.
71	lim = pos, i.e., lim and pos point to the same character in the file.
72	Read access to integer file descriptor fd.
73	Write access to string pointed at by pos.
74	Complement of assumption number 70.
75	Complement of assumption number 32.
76	i < argc, i.e., there are more files to be searched.
77	Write access to character string pointer file.
78	File pointed at by fd can't be opened for reading.
79	Complement of assumption number 78.

TABLE IX

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE C PROGRAM FASTFIND, CASE 3

Object	Assumptions
A.1	1
A.2	4, 5
A.3	6
A.4	10, 11
A.5	13, 14, 15
A.6	16
A.7	15, 17, 18, 19
A.8	6
A.9	15, 17, 20, 21, 22
A.10	16, 17
A.11	16, 17
A.12	18
A.13	12, 17, 23
A.14	24, 25, 26
A.15	17, 24, 27
A.16	12, 28, 29, 30, 31
A.17	17, 18, 27, 31
A.18	4, 32, 33, 34
A.19	35
A.20	4, 33, 34, 75
A.21	24, 25, 76
A.22	11, 24, 77
A.23	35, 61, 72, 78
A.24	61
A.25	35, 61, 72, 79
A.26	72
B.1	36
B.2	37
B.3	38, 39, 40, 41
B.4	38, 42, 43, 44, 45, 46, 47, 48
B.5	38, 39, 45
B.6	38, 46, 48
B.7	59
B.8	38, 39
B.9	70
C.1	37, 55
C.2	39, 41, 55
C.3	38, 40, 70, 71, 72, 73
D.1	38, 49, 50, 51
D.2	12, 29, 30, 52
D.3	41, 53, 54, 55, 56
D.4	31, 51, 57
D.5	58, 59

TABLE IX (Continued)

Object	Assumptions
E.1	33, 60
E.2	61
E.3	43, 62
E.4	38, 39, 41, 55, 63
E.5	38, 54, 64, 65
E.6	41, 53, 54, 55, 56
E.7	38, 53, 66
E.8	41, 53, 54, 55, 66
E.9	65
E.10	43, 68
E.11	38, 39, 48, 69, 70
E.12	43
F.1	2
F.2	2, 3
G.1	7, 8
G.2	7
G.3	9

TABLE X

ASSUMPTIONS FOR THE C PROGRAM
MAIL, CASE 1

Asmp. Number	Assumption
1	There exists an structure of type address with five character string fields (name, street, city, state, and zip) and two pointers of type address (next and prior).
2	Write access is required to pointer called start of type address.
3	Write access is required to pointer called last of type address.
4	Read access is required to pointer called start of type address.
5	Read access is required to pointer called last of type address.
6	There exists a function called menu_select().
7	Integer value returned by function menu_select equals 1.
8	There exists a function called enter().
9	Integer value returned by function menu_select equals 2.
10	There exists a function called delete().
11	Integer value returned by function menu_select equals 3.
12	There exists a function called list().
13	Integer value returned by function menu_select equals 4.
14	There exists a function called search().
15	Integer value returned by function menu_select equals 5.
16	There exists a function called save().
17	Integer value returned by function menu_select equals 6.
18	There exists a function called load().
19	Integer value returned by function menu_select equals 7.
20	There exists a function called inputs() with three parameters.
21	The first and second parameters of the function inputs() have read and write access. The third parameter has only read access.
22	There exists a function called dls_store() with two parameters.
23	Both of the parameters for the dls_store function have read and write access.

TABLE X (Continued)

Asmp. Number	Assumption
24	Function dls_store() returns the address of the first element of a doubly-linked list of structures of type address.
25	There exists a function called find() with one parameter.
26	The parameter for the function called find() has read access only.
27	Function find() returns the address of the structure holding a string that matches the input parameter or NULL if there is no match.
28	There exists a function called display() with one parameter.
29	The parameter for the function display() has read access only.
30	Function menu_select() returns an integer.

TABLE XI

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE C PROGRAM MAIL, CASE 1

Object	Assumptions
A.1	1, 2, 3, 5
A.2	6, 7, 8, 30
A.3	6, 9, 10, 30
A.4	6, 11, 12, 30
A.5	6, 13, 14, 30
A.6	6, 15, 16, 30
A.7	6, 17, 18, 30
A.8	6, 19, 30
B.0	30
C.0	1, 2, 4, 20, 21, 22, 23, 24
D.0	1, 21
E.0	1, 2, 3, 4, 5, 23, 24
F.0	1, 2, 3, 4, 5, 25, 26, 27
G.0	1, 26, 27
H.0	1, 4, 28, 29
I.0	1, 29
J.0	1, 25, 26, 27, 28, 29
K.0	1, 4
L.0	1, 2, 3, 4, 5

TABLE XII

ASSUMPTIONS FOR THE C PROGRAM
MAIL, CASE 2

Asmp. Number	Assumption
1	There exists an structure of type address with five character string fields (name, street, city, state, and zip) and two pointers of type address (next and prior).
2	Write access is required to pointer called start of type address.
3	Write access is required to pointer called last of type address.
4	Read access is required to pointer called start of type address.
5	Read access is required to pointer called last of type address.
6	There exists a function called menu_select().
7	Function menu_select() returns an integer.
8	Integer value returned by function menu_select equals 1.
9	There exists a module called module_1.
10	Integer value returned by function menu_select equals 2.
11	Integer value returned by function menu_select equals 3.
12	Integer value returned by function menu_select equals 4.
13	Integer value returned by function menu_select equals 5.
14	There exists a function called save().
15	Integer value returned by function menu_select equals 6.
16	There exists a function called load().
17	Integer value returned by function menu_select equals 7.
18	There exists a function called find() with one parameter.
19	The parameter for the function called find() has read access only.
20	Function find() returns the address of the structure holding a string that matches the input parameter or NULL if there is no match.
21	There exists a function called display() with one parameter.
22	The parameter for the function display() has read access only.

TABLE XII (Continued)

Asmp. Number	Assumption
23	Write access to character string s.
24	Write access to pointer info of type address.
25	Read access to pointer info of type address.
26	Pointer info points to the structure to be deleted or NULL if the structure was not found.
27	Pointer info points to the first structure in the list.
28	Read access to the field next in structure pointed at by info.
29	List is not empty.
30	List is empty.
31	Write access to the field prior in structure pointed at by start.
32	Complement of assumption number 26.
33	Read access to the field prior in structure pointed at by info.
34	Write access to the field next in structure pointed at by the field prior in the structure pointed at by info.
35	Structure pointed at by info is not the last structure in the list.
36	Write access to the field prior in structure pointed at by the field next in the structure pointed at by info.
37	Pointer info points to the last structure in the list.
38	Pointer info points to the structure being presently looked at in the list or NULL if there are no more structures in the list.
39	Pointer info points to the structure to be displayed or NULL if the structure was not found.

TABLE XIII

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE C PROGRAM MAIL, CASE 2

Object	Assumptions
A.1	1, 2, 3, 5
A.2	6, 7, 8, 9
A.3	6, 7, 10
A.4	6, 7, 11
A.5	6, 7, 12
A.6	6, 7, 13, 14
A.7	6, 7, 15, 16
A.8	6, 7, 17
B.0	7
CDE.0	1, 2, 3, 4, 5
F.1	23
F.2	1, 18, 19, 20, 24
F.3	1, 25, 26
F.4	1, 4, 25, 27
F.5	1, 2, 28
F.6	1, 4, 29
F.7	1, 31
F.8	30
F.9	1, 2
F.10	32
F.11	1, 28, 33, 34
F.12	1, 5, 25, 35
F.13	1, 28, 33, 36
F.14	1, 37
F.15	1, 3, 33
F.16	1, 25
G.0	1, 19, 20
H.1	1, 4, 24
H.2	1, 25, 38
H.3	1, 21, 22, 25
H.4	1, 24, 28
I.0	1, 22
J.1	23
J.2	1, 19, 20, 21, 24, 39
J.3	1, 39
J.4	1, 21, 22, 25
K.0	1, 4
L.0	1, 2, 3, 4, 5

TABLE XIV

ASSUMPTIONS FOR THE C PROGRAM
MAIL, CASE 3

Asmp. Number	Assumption
1	Read access to pointer start of type address.
2	Write access to pointer start of type address.
3	Read access to pointer last of type address.
4	Write access to pointer last of type address.
5	Write access to string s.
6	Read access to string s.
7	Write access to integer c.
8	Read access to integer c.
9	c is in the range c < 1 or c > 7.
10	c = 1.
11	Structure of type address exists.
12	Write access to pointer info of type address.
13	Memory space for pointer info is not available.
14	Read access to pointer info of type address.
15	Write access to integer count.
16	Write access to string p.
17	Read access to string p.
18	Length of string p is larger than value in count.
19	Write access to field name in structure pointed at by info.
20	Read access to field name in structure pointed at by info.
21	Length of field name in structure pointed at by info is 0.
22	Write access to field street in structure pointed at by info.
23	Write access to field city in structure pointed at by info.
24	Write access to field state in structure pointed at by info.
25	Write access to field zip in structure pointed at by info.
26	last = NULL and list is empty.
27	Write access to field next in structure pointed at by info.
28	Write access to field prior in structure pointed at by info.
29	Write access to pointer p of type address.
30	Write access to pointer old of type address.
31	Pointer p points to structure being presently looked at in the list or NULL if there are no more structures in the list.

TABLE XIV (Continued)

Asmp. Number	Assumption
32	Read access to pointer p of type address.
33	Read access to field name in structure pointed at by info.
34	Read access to field name in structure pointed at by p.
35	Field name in structure pointed at by p precedes alphabetically the field name in structure pointed at by info.
36	Read access to field next in structure pointed at by p.
37	Complement of assumption number 35.
38	Read access to field prior in structure pointed at by p.
39	Structure pointed at by p is not the first one in the list.
40	Write access to field next in structure pointed at by the field prior in the structure pointed at by p.
41	Write access to field prior in structure pointed at by p.
42	Read access to pointer old of type address.
43	c = 2.
44	Pointer info points to the structure being presently looked at in the list or NULL if there are no more structures in the list.
45	String in s is the same as the string in the field name in the structure pointed at by info.
46	Read access to field next in structure pointed at by info.
47	Pointer info points to the structure to be deleted or NULL if the structure was not found.
48	Pointer info points to the first structure in the list.
49	List is not empty.
50	Write access to field prior in structure pointed at by start.
51	Complement of assumption number 49.
52	Complement of assumption number 47.
53	Read access to field prior in structure pointed at by info.
54	Write access to field next in structure pointed at by the field prior in the structure pointed at by info.
55	Structure pointed at by info is not the last structure in the list.

TABLE XIV (Continued)

Asmp. Number	Assumption
56	Write access to field prior in structure pointed at by the field next in structure pointed at by info.
57	Pointer info points to the last structure in the list.
58	Read access to field street in structure pointed at by info.
59	Read access to field city in structure pointed at by info.
60	Read access to field state in structure pointed at by info.
61	Read access to field zip in structure pointed at by info.
62	Pointer info points to the structure to be displayed or NULL if the structure was not found.
63	c = 3.
64	c = 4.
65	c = 5.
66	c = 6.
67	c = 7.
68	Write access to field next in structure pointed at by old.
69	File mail_list exists.
70	File mail_list can be opened for writing.
71	Write access to file pointer fp.
72	Read access to file pointer fp.
73	File mail_list can be opened for reading.
74	Pointer start points to the present structure being looked at or NULL if there are no more structures in the list.
75	Read access to field next in structure pointed at by start.
76	Memory space for pointer start is not available.
77	End of the file pointed by fp has not been reached.
78	Memory space for pointer pointed by the field next in the structure pointed at by info, is not available.
79	Read access to pointer temp of type address.
80	Write access to pointer temp of type address.
81	Write access to field next in the structure pointed at by temp.

TABLE XV

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE C PROGRAM MAIL, CASE 3

Object	Assumptions
A.1	2, 3, 4, 11
A.2	8, 10
A.3	8, 43
A.4	8, 63
A.5	8, 64
A.6	8, 65
A.7	8, 66
A.8	8, 67
B.1	5
B.2	6, 7
B.3	8, 9
C.1	11, 12
C.2	11, 13, 14
C.3	11, 15, 19
C.4	11, 20, 21
C.5	11, 15, 22
C.6	11, 15, 23
C.7	11, 15, 24
C.8	11, 15, 25
C.9	2, 11
D.1	16
D.2	17, 18
D.3	17, 18
E.1	3, 11, 26
E.2	11, 27
E.3	11, 28
E.4	4, 11, 14
E.5	11, 14
E.6	1, 11, 29
E.7	30
E.8	31, 32
E.9	11, 31, 33, 34, 35
E.10	30, 32
E.11	11, 29, 36
E.12	37
E.13	11, 38, 39
E.14	11, 14, 38, 40
E.15	11, 27, 32
E.16	11, 28, 38
E.17	11, 14, 41
E.18	1, 11
E.19	11, 27, 32
E.20	11, 28

TABLE XV (Continued)

Object	Assumptions
E.21	11, 14, 41
E.22	11, 14
E.23	11, 14, 68
E.24	11, 27
E.25	11, 28, 42
E.26	4, 11, 14
E.27	1, 11
F.1	5
F.2	11, 12
F.3	11, 14, 47
F.4	1, 11, 14, 48
F.5	2, 11, 46
F.6	1, 11, 49
F.7	1, 11, 50
F.8	51
F.9	2, 11
F.10	52
F.11	11, 46, 53, 54
F.12	3, 11, 14, 55
F.13	11, 46, 53, 56
F.14	11, 57
F.15	4, 11, 53
F.16	11, 14
G.1	1, 11, 12
G.2	11, 14, 44
G.3	6, 11, 33, 45
G.4	11, 14
G.5	11, 12, 46
H.1	1, 11, 12
H.2	11, 14, 44
H.3	11, 14
H.4	11, 12, 46
I.1	11, 33
I.2	11, 58
I.3	11, 59
I.4	11, 60
I.5	11, 61
J.1	5
J.2	11, 12, 62
J.3	11, 62
J.4	11, 14
K.1	69, 70, 71, 72
K.2	1, 11, 12
K.3	11, 14, 44
K.4	11, 14, 72
K.5	11, 12, 46

TABLE XV (Continued)

Object	Assumptions
K.6	72
L.1	69, 71, 72, 73
L.2	1, 11, 74
L.3	11, 12, 75
L.4	11, 14
L.5	2, 11, 14
L.6	1, 11
L.7	1, 11, 76
L.8	1, 11, 12
L.9	72, 77
L.10	11, 12, 72, 77
L.11	11, 27
L.12	11, 46, 78
L.13	11, 28, 79
L.14	11, 14, 80
L.15	11, 12, 46
L.16	11, 81
L.17	4, 11, 79
L.18	11, 50
L.19	72

TABLE XVI

ASSUMPTIONS FOR THE C PROGRAM EDITOR, CASE 1

Asmp. Number	Assumption
1	There exists an structure of type line with one character string field (text), one integer field (num), and two pointers of type line (next and prior).
2	Write access is required to pointer called start of type line.
3	Write access is required to pointer called last of type line.
4	Read access is required to pointer called start of type line.
5	Read access is required to pointer called last of type line.
6	There exists a function called menu_select().
7	Function menu_select() returns an integer.
8	Character choice equals 1.
9	There exists a function called enter() with one parameter.
10	The parameter for the function enter() has read and write access.
11	Character choice equals 2.
12	There exists a function called delete().
13	Character choice equals 3.
14	There exists a function called list().
15	Character choice equals 4.
16	There exists a function called save() with one parameter.
17	Character choice equals 5.
18	There exists a function called load() with one parameter.
19	Character choice equals 6.
20	There exists a function called patchup().
21	Both parameters of the patchup() function have read access only.
22	There exists a function called dls_store() with one parameter.
23	The parameter for the dls_store() function has read and write access.
24	Function dls_store() returns the address of the first element of a doubly-linked list of structures of type line.
25	There exists a function called find() with one parameter.

TABLE XVI (Continued)

Asmp. Number	Assumption
26	The parameter for the function find() has read access only.
27	Function find() returns the address of the structure holding the line number that matches the input parameter or NULL if there is no match.
28	The parameter for the function save() has read access only.
29	The parameter for the function load() has read access only.
30	Each line stored in the field, text, in the structure type line is terminated with a carriage return (\r) character followed by a new line character (\n).
31	Write access to integer variable linenum.
32	Read access to integer variable argc.
33	Integer variable argc equals 2, i.e., filename present in command line.
34	Read access to character string argv.
35	Write access to character choice.
36	Read access to character choice.
37	Read access to integer variable linenum.
38	Write access to character string s.
39	Read access to character string s.
40	Write access to character string fname.
41	Read access to character string fname.

TABLE XVII

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE C PROGRAM EDITOR, CASE 1

Object	Assumptions
A.1	1, 2
A.2	1, 3
A.3	32, 33
A.4	18, 29, 34
A.5	6, 7, 35
A.6	8, 36
A.7	38
A.8	31, 39
A.9	9, 10, 31, 37
A.10	11, 12, 36
A.11	13, 14, 36
A.12	15, 36
A.13	40
A.14	16, 28, 41
A.15	17, 36
A.16	40
A.17	18, 29, 41
A.18	19, 36
B.0	7
C.0	1, 2, 10, 20, 21, 22, 23, 24, 25, 26, 27
D.0	1, 21, 25, 26, 27
E.0	1, 2, 3, 4, 5, 23, 24
F.0	1, 2, 3, 4, 5, 20, 21, 25, 26, 27
G.0	1, 4, 26, 27
H.0	1, 4
I.0	1, 4, 28, 30
J.0	1, 2, 3, 4, 29, 30

TABLE XVIII

ASSUMPTIONS FOR THE C PROGRAM EDITOR, CASE 2

Asmp. Number	Assumption
1	There exists an structure of type line with one character string field (text), one integer field (num), and two pointers of type line (next and prior).
2	Write access is required to pointer called start of type line.
3	Write access is required to pointer called last of type line.
4	Read access is required to pointer called start of type line.
5	Read access is required to pointer called last of type line.
6	There exists a function called menu_select().
7	Function menu_select() returns an integer.
8	Character choice equals 1.
9	Character choice equals 2.
10	Character choice equals 3.
11	There exists a function called list().
12	Character choice equals 4.
13	There exists a function called save() with one parameter.
14	The parameter for the function save() has read access only.
15	Character choice equals 5.
16	There exists a function called load() with one parameter.
17	The parameter for the function load() has read access only.
18	Character choice equals 6.
19	There exists a function called dls_store() with one parameter.
20	The parameter for the dls_store() function has read and write access.
21	Function dls_store() returns the address of the first element of a doubly-linked list of structures of type line.
22	Each line stored in the field, text, in the structure type line is terminated with a carriage return (\r) character followed by a new line character (\n).
23	Write access to integer variable linenum.
24	Read access to integer variable argc.

TABLE XVIII (Continued)

Asmp. Number	Assumption
25	Integer variable argc equals 2, i.e., filename present in command line.
26	Read access to character string argv.
27	Write access to character choice.
28	Read access to character choice.
29	Read access to integer variable linenum.
30	Write access to character string s.
31	Read access to character string s.
32	Write access to character string fname.
33	Read access to character string fname.
34	There exists a module called module_1 with three parameters.
35	All parameters in module module_1 have read access only.
36	Module module_1 is in patchup mode.
37	Module module_1 is in find mode.
38	Module module_1 returns the address of the structure holding the line number that matches the input parameter or NULL if there is no match when module_1 is in find mode.
39	Write access to pointer info of type line.
40	Memory space for pointer info is not available.
41	Read access to pointer info of type line.
42	Write access to field text in structure pointed at by info.
43	Read access to field text in structure pointed at by info.
44	The first character in the field text in the structure pointed at by info is not the line terminator ('\0').
45	Pointer info points to the structure in the list whose field num has the same value as linenum.
46	Write access to integer variable increment.
47	Complement of assumption number 44.
48	Pointer info points to the structure to be deleted or NULL if the structure was not found.
49	Pointer info points to the first structure in the list.
50	Read access to field next in structure pointed at by info.
51	List is not empty.
52	Write access to field prior in structure pointed at by start.
53	Complement of assumption number 51.
54	Complement of assumption number 48.

TABLE XVIII (Continued)

Asmp. Number	Assumption
55	Read access to field prior in structure pointed at by info.
56	Write access to field next in structure pointed at by the field prior in the structure pointed at by info.
57	Structure pointed at by info is not the last structure in the list.
58	Write access to field prior in structure pointed at by the field next in structure pointed at by info.
59	Pointer info points to the last structure in the list.

TABLE XIX

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE C PROGRAM EDITOR, CASE 2

Object	Assumptions
A.1	1, 2
A.2	1, 3
A.3	24, 25
A.4	16, 17, 26
A.5	6, 7, 27
A.6	8, 28
A.7	30
A.8	23, 31
A.9	23, 29
A.10	9, 28
A.11	10, 11, 28
A.12	12, 28
A.13	32
A.14	13, 14, 33
A.15	15, 28
A.16	32
A.17	16, 17, 33
A.18	18, 28
B.0	7
C.1	1, 39
C.2	1, 40, 41
C.3	29
C.4	1, 42
C.5	1, 29, 42
C.6	43, 44
C.7	29, 34, 35, 37, 38
C.8	29, 34, 35, 36
C.9	43, 44
C.10	1, 2, 19, 20, 21, 41
C.11	43, 47
C.12	23, 29
DG.0	1, 4, 35, 38
E.0	1, 2, 3, 4, 5, 20, 21
F.1	30
F.2	23, 31
F.3	1, 34, 35, 37, 38, 39
F.4	1, 41, 48
F.5	1, 4, 41, 49
F.6	1, 2, 50
F.7	1, 4, 51
F.8	1, 52
F.9	1, 4, 53
F.10	1, 2

TABLE XIX (Continued)

Object	Assumptions
F.11	1, 41, 54
F.12	1, 50, 55, 56
F.13	1, 5, 41, 57
F.14	1, 50, 55, 58
F.15	1, 5, 41, 59
F.16	1, 3, 55
F.17	1, 41
F.18	46
H.0	1, 4
I.0	1, 4, 14, 22
J.0	1, 2, 3, 4, 17, 22

TABLE XX

ASSUMPTIONS FOR THE C PROGRAM EDITOR, CASE 3

Asmp. Number	Assumption
1	Write access to integer variable linenum.
2	Write access to pointer start of type line.
3	Write access to pointer last of type line.
4	argc = 2, i.e., filename present in command line.
5	Read access to integer variable argc.
6	Read access to character string argv.
7	Write access to character choice.
8	Write access to string s.
9	Read access to string s.
10	Write access to integer c.
11	Read access to integer c.
12	c is in the range c < 1 or c > 6.
13	choice = 1.
14	Read access to integer variable linenum.
15	choice = 2.
16	choice = 3.
17	choice = 4.
18	Write access to character string fname.
19	choice = 5.
20	choice = 6.
21	Structure of type line exists.
22	Write access to pointer info of type line.
23	Memory space for pointer info is not available.
24	Read access to pointer info of type line.
25	Write access to field text in structure pointed at by info.
26	Read access to field text in structure pointed at by info.
27	The first character in the field text in the structure pointed at by info is not the line terminator ('\0').
28	Pointer info points to the structure being presently looked at in the list or NULL if there are no more structures in the list.
29	Read access to field text in structure pointed at by info.
30	The value of the integer linenum is the same as the value in the field num in the structure pointed at by info.
31	Read access to field next in structure pointed at by info.

TABLE XX (Continued)

Asmp. Number	Assumption
32	Pointer info points to the structure in the list whose field num has the same value as linenum.
33	Write access to pointer i of type line.
34	Pointer i points to the structure whose field num needs to be incremented by one.
35	Read access to pointer i of type line.
36	Read access to field num in the structure pointed at by i.
37	Write access to field num in the structure pointed at by i.
38	Write access to integer variable increment.
39	Read access to integer variable increment.
40	Read access to field next in the structure pointed at by i.
41	Read access to pointer last of type line.
42	last = NULL and list is empty.
43	Write access to field next in structure pointed at by info.
44	Write access to field prior in structure pointed at by info.
45	Read access to pointer start of type line.
46	Write access to pointer p of type line.
47	Write access to pointer old of type line.
48	Pointer p points to structure being presently looked at in the list or NULL if there are no more structures in the list.
49	Read access to pointer p of type line.
50	Read access to field num in structure pointed at by info.
51	Read access to field num in structure pointed at by p.
52	Field num in structure pointed at by p is smaller than the field num in structure pointed at by info.
53	Read access to field next in structure pointed at by p.
54	Complement of assumption number 52.
55	Read access to field prior in structure pointed at by p.
56	Structure pointed at by p is not the first one in the list.
57	Write access to field next in structure pointed at by the field prior in the structure pointed at by p.
58	Write access to field prior in structure pointed at by p.

TABLE XX (Continued)

Asmp. Number	Assumption
59	Write access to field next in structure pointed at by old.
60	Read access to pointer old of type address.
61	Complement of assumption number 27.
62	Pointer info points to the structure to be deleted or NULL if the structure was not found.
63	Pointer info points to the first structure in the list.
64	List is not empty.
65	Write access to field prior in structure pointed at by start.
66	Complement of assumption number 64.
67	Complement of assumption number 62.
68	Read access to field prior in structure pointed at by info.
69	Write access to field next in structure pointed at by the field prior in the structure pointed at by info.
70	Structure pointed at by info is not the last structure in the list.
71	Write access to field prior in structure pointed at by the field next in structure pointed at by info.
72	Pointer info points to the last structure in the list.
73	Filename fname can be opened for writing.
74	Write access to file pointer fp.
75	Read access to character string fname.
76	Write access to character string pointer p.
77	The character pointed at by p is not the null terminator ('\0').
78	Read access to character string pointer p.
79	Read access to the character pointed at by p.
80	Read access to file pointer fp.
81	Filename fname can be opened for reading.
82	Pointer start points to the present structure being looked at or NULL if there are no more structures in the list.
83	Write access to pointer temp of type line.
84	Read access to pointer temp of type line.
85	Read access to field next in structure pointed at by start.
86	Write access to register variable size.
87	Read access to register variable size.
88	Memory space for pointer start is not available.
89	Write access to register variable lnct.

TABLE XX (Continued)

Asmp. Number	Assumption
90	Write access to character pointed at by p.
91	End of the file pointed by fp has not been reached.
92	The character pointed at by p is not a printable character.
93	The character pointed at by p is not a carriage return ('\r').
94	Write access to field num in structure pointed at by info.
95	Read access to register variable lnct.
96	Memory space for pointer pointed by the field next in the structure pointed at by info, is not available.
97	Read access to pointer temp of type address.
98	Write access to pointer temp of type address.
99	Write access to field next in the structure pointed at by temp.
100	Read access to character choice.

TABLE XXI

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE C PROGRAM EDITOR, CASE 3

Object	Assumptions
A.1	2, 21
A.2	3, 21
A.3	4, 5, 21
A.4	6
A.5	7, 11
A.6	13, 100
A.7	8
A.8	1, 9
A.9	1, 14
A.10	15, 100
A.11	16, 100
A.12	17, 100
A.13	18
A.14	75
A.15	19, 100
A.16	18
A.17	75
A.18	20, 100
B.1	8
B.2	9, 10
B.3	11, 12
C.1	21, 22
C.2	21, 23, 24
C.3	14
C.4	21, 25
C.5	14, 21, 25
C.6	26, 27
C.7	14, 21, 24, 32
C.8	14, 38
C.9	26, 27
C.10	2, 21, 24
C.11	26, 61
C.12	1, 14
D.1	14, 21, 33
D.2	21, 34, 35
D.3	21, 36, 37, 39
D.4	21, 35, 40
E.1	21, 41, 42
E.2	21, 43
E.3	21, 44
E.4	3, 21, 24
E.5	21, 24
E.6	21, 45, 46

TABLE XXI (Continued)

Object	Assumptions
E.7	21, 47
E.8	21, 48, 49
E.9	21, 48, 50, 51, 52
E.10	21, 47, 49
E.11	21, 46, 53
E.12	21, 48, 50, 51, 54
E.13	21, 55, 56
E.14	21, 24, 55, 57
E.15	21, 43, 49
E.16	21, 44, 55
E.17	21, 24, 58
E.18	21, 45
E.19	21, 43, 49
E.20	21, 44
E.21	21, 24, 58
E.22	21, 24
E.23	21, 24, 59
E.24	21, 43
E.25	21, 44, 60
E.26	3, 21, 24
E.27	21, 45
F.1	8
F.2	1, 9
F.3	1, 21, 22
F.4	21, 24, 62
F.5	21, 24, 45, 63
F.6	2, 21, 31
F.7	21, 45, 64
F.8	21, 65
F.9	21, 45, 66
F.10	2, 21
F.11	21, 24, 67
F.12	21, 31, 68, 69
F.13	21, 24, 41, 70
F.14	21, 31, 68, 71
F.15	21, 24, 41, 72
F.16	3, 21, 68
F.17	21, 24
F.18	38
G.1	1, 21, 22
G.2	21, 24, 28
G.3	14, 21, 29, 30
G.4	21, 24
G.5	21, 22, 31
H.1	21, 22, 45
H.2	21, 24, 48

TABLE XXI (Continued)

Object	Assumptions
H.3	21, 29, 50
H.4	21, 22, 31
I.1	73, 74, 75
I.2	21, 22, 45
I.3	21, 24, 28
I.4	21, 29, 76
I.5	77, 79
I.6	76, 78, 79
I.7	80
I.8	80
I.9	21, 22, 31
I.10	80
J.1	74, 75, 81
J.2	21, 45, 82
J.3	21, 45, 83
J.4	2, 21, 85
J.5	21, 84
J.6	21, 86
J.7	2, 21, 87
J.8	88
J.9	21, 22, 45
J.10	21, 29, 76
J.11	89
J.12	79, 80, 90, 91
J.13	79, 92
J.14	76, 78
J.15	79, 80, 90, 93
J.16	76, 78
J.17	80
J.18	90
J.19	89, 94, 95
J.20	21, 43, 87
J.21	21, 31, 96
J.22	21, 44, 97
J.23	21, 24, 98
J.24	21, 22, 31
J.25	21, 29, 76
J.26	21, 99
J.27	3, 21, 97
J.28	21, 24
J.29	21, 65
J.30	80

TABLE XXII

ASSUMPTIONS FOR THE ADA PROGRAM
INTLIST, CASE 1

Asmp. Number	Assumption
1	Read access to pointer Head of type link.
2	There exists a record of type List with an integer field called Value and a pointer to the next record of type link called Next.
3	Read access to the field Next in the pointer Head.
4	Write access to pointer Head of type link.
5	Read access to pointer Free of type link.
6	Write access to pointer Free of type link.
7	Write access to field Next in pointer Free.
8	Write access to pointer Tail of type link.
9	Write access to integer variable Number.
10	Read access to integer variable Number.
11	Integer variable Number equals -1.
12	Read access to pointer Tail of type link.
13	Write access to field Next in pointer Tail.
14	Read access to field Next in pointer Tail.
15	Read access to integer variable I.
16	Write access to integer variable I.
17	There exists a procedure called Insert_At_Head with one parameter.
18	The parameter in procedure Insert_At_Head has read access only.
19	There exists a procedure called Insert_At_Tail with one parameter.
20	The parameter in procedure Insert_At_Tail has read access only.
21	There exists a function called List_Length.
22	Function List_Length returns the number of elements in the list, i.e., the length of the list.
23	There exists a function called Value_At_Position with one parameter.
24	The parameter in function Value_At_Position has read access only.
25	Function Value_At_Position returns the number stored in the field Value in the position determined by the input parameter.
26	There exists a procedure called Reclaim with one parameter.
27	The parameter in procedure Reclaim has read access only.
28	There exists a function called Alloc with one parameter.

TABLE XXII (Continued)

Asmp. Number	Assumption
29	The parameter in function Alloc has read access only.
30	Function Alloc returns a pointer of type Link to the newly created record that contains the new value and a pointer to the next record.

TABLE XXIII

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE ADA PROGRAM INTLIST, CASE 1

Object	Assumptions
A.1	9
A.2	10, 11
A.3	10, 17, 18
A.4	10, 19, 20
A.5	15, 16, 21, 22
A.6	15, 23, 24, 25
A.7	15, 16, 21, 22
A.8	15, 23, 24, 25
B.0	2, 5, 6, 7, 27
C.0	2, 5, 7, 29, 30
D.0	1, 2, 3, 4, 8, 26, 27
E.0	1, 2, 4, 8, 12, 18, 28, 29, 30
F.0	1, 2, 4, 8, 13, 14, 20, 28, 29, 30
G.0	1, 2, 3, 4, 8, 26, 27
H.0	1, 2, 4, 8, 12, 13, 26, 27
I.0	1, 2, 24, 25
J.0	1, 2, 22

TABLE XXIV

ASSUMPTIONS FOR THE ADA PROGRAM
INTLIST, CASE 2

Asmp. Number	Assumption
1	Read access to pointer Head of type link.
2	There exists a record of type List with an integer field called Value and a pointer to the next record of type link called Next.
3	Read access to the field Next in the pointer Head.
4	Write access to pointer Head of type link.
5	Read access to pointer Free of type link.
6	Write access to pointer Free of type link.
7	Write access to field Next in pointer Free.
8	Write access to pointer Tail of type link.
9	Write access to integer variable Number.
10	Read access to integer variable Number.
11	Integer variable Number equals -1.
12	Read access to pointer Tail of type link.
13	Write access to field Next in pointer Tail.
14	Read access to field Next in pointer Tail.
15	Read access to integer variable I.
16	Write access to integer variable I.
17	There exists a function called List_Length.
18	Function List_Length returns the number of elements in the list, i.e., the length of the list.
19	There exists a function called Value_At_Position with one parameter.
20	The parameter in function Value_At_Position has read access only.
21	Function Value_At_Position returns the number stored in the field Value in the position determined by the input parameter.
22	There exists a procedure called Reclaim with one parameter.
23	The parameter in procedure Reclaim has read access only.
24	There exists a module called module_1 with two parameters.
25	The first parameter in module_1 has read access and the second, called mode selector, has also read access.
26	The mode selector in module_1 is in Insert_At_Head mode.
27	The mode selector in module_1 is in Insert_At_Tail mode.
28	Write access to pointer P of type link.

TABLE XXIV (Continued)

Asmp. Number	Assumption
29	Read access to the field Next in the pointer Head.
30	Write access to pointer Head of type link.
31	Read access to pointer P of type link.
32	Pointer Head is null, i.e., list is empty.
33	Write access to pointer Tail of type Link.
34	Read access to pointer Tail of type link.
35	Pointers Head and Tail are equal, i.e., only one record in list.
36	Complement of assumption 35.
37	Read access to field Next in pointer P.
38	The record pointed by the field Next in pointer P is not the same record pointed at by pointer Tail.
39	Write access to field Next in pointer Tail.

TABLE XXV

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE ADA PROGRAM INTLIST, CASE 2

Object	Assumptions
A.1	9
A.2	10, 11
A.3	10, 24, 25, 26
A.4	10, 24, 25, 27
A.5	15, 16, 17, 18
A.6	15, 19, 20, 21
A.7	15, 16, 17, 18
A.8	15, 19, 20, 21
B.0	2, 5, 6, 7, 23
CEF.0	1, 2, 4, 5, 7, 8, 12, 13, 14, 25
D.0	1, 2, 3, 4, 8, 22, 23
G.1	1, 28
G.2	2, 29, 30
G.3	22, 23, 31
G.4	1, 32
G.5	33
H.1	1, 28
H.2	1, 34, 35
H.3	30
H.4	33
H.5	1, 34, 36
H.6	2, 34, 37, 38
H.7	2, 28, 37
H.8	22, 23, 34
H.9	31, 33
H.10	2, 39
I.0	1, 2, 20, 21
J.0	1, 2, 18

TABLE XXVI

ASSUMPTIONS FOR THE ADA PROGRAM
INTLIST, CASE 3

Asmp. Number	Assumption
1	Read access to pointer Head of type link.
2	Pointer Head is not null, i.e., points to the first record in the list.
3	Write access to pointer P of type link.
4	There exists a record of type List with an integer field called Value and a pointer to the next record of type link called Next.
5	Read access to the field Next in the pointer Head.
6	Write access to pointer Head of type link.
7	Read access to pointer P of type link.
8	Read access to pointer Free of type link.
9	Pointer Free is null, i.e., no records in free list.
10	Write access to pointer Free of type link.
11	Write access to field Next in pointer Free.
12	Complement of assumption 9.
13	Write access to field Next in pointer P.
14	Write access to pointer Tail of type link.
15	Write access to integer variable Number.
16	Read access to integer variable Number.
17	Integer variable Number equals -1.
18	Write access to integer variable Value.
19	Read access to integer variable Value.
20	Pointer Tail is null, i.e., number list is empty.
21	Read access to pointer Tail of type link.
22	Pointer P is null, i.e., list is empty.
23	Complement of assumption 22.
24	Write access to field Value in pointer P.
25	Write access to field Value in pointer Initial_Value.
26	Write access to field Next in pointer Initial_Value.
27	Read access to field Value in pointer Initial_Value.
28	Read access to field Next in pointer Initial_Value.
29	Complement of assumption 2.
30	Write access to field Next in pointer Tail.
31	Read access to field Next in pointer Tail.
32	Read access to integer variable I.
33	Write access to integer variable I.
34	Read access to natural variable LEN.

TABLE XXVI (Continued)

Asmp. Number	Assumption
35	Write access to natural variable LEN.
36	Read access to field Next in pointer P.
37	Read access to positive variable Pos.
38	Write access to positive variable Pos.
39	Read access to field Value in pointer P.
40	Pointers Head and Tail are equal, i.e., only one record in list.
41	Complement of assumption 40.
42	The record pointed by the field Next in pointer P is not the same record pointed at by pointer Tail.

TABLE XXVII

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE ADA PROGRAM INTLIST, CASE 3

Object	Assumptions
A.1	15
A.2	16, 17
A.3	16, 18
A.4	16, 18
A.5	32, 33, 34
A.6	32, 38, 39
A.7	32, 33, 34
A.8	32, 38, 39
B.1	8, 9
B.2	7, 10
B.3	4, 11
B.4	9, 12
B.5	4, 8, 13
B.6	7, 10
C.1	3, 8
C.2	7, 22
C.3	3, 4
C.4	7, 23
C.5	4, 10, 11
C.6	4, 13, 24, 27, 28
C.7	7
D.1	1, 2
D.2	1, 3
D.3	4, 5, 6
D.4	7
D.5	14
E.1	1, 4, 6, 19, 25, 2(
E.2	20, 21
E.3	1, 14
F.1	1, 29
F.2	4, 6, 19, 25, 26
F.3	1, 14
F.4	1, 2
F.5	4, 19, 25, 26, 30
F.6	4, 14, 31
G.1	1, 3
G.2	4, 5, 6
G.3	7
G.4	1, 29
G.5	14
H.1	1, 3
H.2	1, 21, 40
H.3	6

TABLE XXVII (Continued)

Object	Assumptions
H.4	14
H.5	1, 21, 41
H.6	4, 21, 36, 42
H.7	3, 4, 36
H.8	3, 21
H.9	7, 14
H.10	4, 30
I.1	1, 3
I.2	32, 33, 37
I.3	3, 4, 36
I.4	4, 39
J.1	1, 3
J.2	7, 23
J.3	34, 35
J.4	3, 4, 36
J.5	34

TABLE XXVIII

ASSUMPTIONS FOR THE ADA PROGRAM
CALC, CASE 1

Asmp. Number	Assumption
1	Write access to string variable STR.
2	Write access to float variable NUM_VAL.
3	Write access to boolean variable FIRST.
4	Read access to string variable STR.
5	The first character in string variable STR is not "q" or "Q".
6	Read access to boolean variable FIRST.
7	Boolean variable FIRST is not true.
8	Write access to natural variable LEN.
9	The first character in string variable STR is either a digit, a ".", or a "-" and the value of the natural variable LEN is larger than one.
10	Read access to natural variable LEN.
11	Read access to natural variable TOP.
12	Read access to natural constant LIMIT.
13	Write access to natural variable TOP.
14	Write access to array of float STACK.
15	Write access to float variable NUM.
16	Read access to float variable NUM_VAL.
17	Read access to float variable NUM.
18	Complement of assumption 9.
19	Read access to array of float STACK.
20	Exception INVALID_ENTRY has been raised.
21	Exception NUMERIC_ERROR has been raised.
22	There exists a procedure called GET_STRING with two parameters.
23	Both of the parameters in procedure GET_STRING have write access only.
24	There exists a procedure called STR_TO_FLT with three parameters.
25	Two parameters in procedure STR_TO_FLT have read access and the third parameter has write access.
26	There exists a procedure called PUSH with one param.
27	The parameter in procedure PUSH has read access.
28	There exists a procedure called OPERATE with one parameter.
29	The parameter in procedure OPERATE has read access.
30	There exists a procedure called POP with one param.
31	The parameter in procedure POP has write access.
32	There exists a procedure called CLEAR with no parameters.

TABLE XXIX

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE ADA PROGRAM CALC, CASE 1

Object	Assumptions
A.1	4, 5
A.2	6, 7
A.3	1, 8, 22, 23
A.4	3
A.5	4, 9
A.6	2, 4, 10, 24, 25
A.7	16, 26, 27
A.8	4, 18
A.9	4, 28, 29
A.10	20
A.11	21
B.0	26, 27, 29, 30, 31, 32
C.0	11, 12, 13, 14, 17, 27
D.0	11, 13, 15, 19, 31
E.0	13
F.0	23
G.0	25

TABLE XXX

ASSUMPTIONS FOR THE ADA PROGRAM
CALC, CASE 2

Asmp. Number	Assumption
1	Write access to string variable STR.
2	Write access to float variable NUM_VAL.
3	Write access to boolean variable FIRST.
4	Read access to string variable STR.
5	The first character in string variable STR is not "q" or "Q".
6	Read access to boolean variable FIRST.
7	Boolean variable FIRST is not true.
8	Write access to natural variable LEN.
9	The first character in string variable STR is either a digit, a ".", or a "-" and the value of the natural variable LEN is larger than one.
10	Read access to natural variable LEN.
11	Read access to float variable NUM_VAL.
12	Complement of assumption 9.
13	Exception INVALID_ENTRY has been raised.
14	Exception NUMERIC_ERROR has been raised.
15	There exists a procedure called GET_STRING with two parameters.
16	Both of the parameters in procedure GET_STRING have write access only.
17	There exists a procedure called STR_TO_FLT with three parameters.
18	Two parameters in procedure STR_TO_FLT have read access and the third parameter has write access.
19	Read access to float variable X.
20	Write access to string variable STRG.
21	Read access to string variable STRG.
22	The first character in string variable STRG is "+".
23	The first character in string variable STRG is "-".
24	The first character in string variable STRG is "*".
25	The first character in string variable STRG is "/".
26	The first character in string variable STRG is "r" or "R".
27	The first character in string variable STRG is "d" or "D".
28	The first character in string variable STRG is "?".
29	The first character in string variable STRG is "q" or "Q".
30	The first character in string variable STRG is not "+", "-", "*", "/", "r", "R", "d", "D", "?", "q", nor "Q".

TABLE XXX (Continued)

Asmp. Number	Assumption
31	Read access to float variable Y.
32	Write access to float variable Y.
33	There exists a module called module_1 with two parameters.
34	The first parameter in module_1 has read and write access and the second, called mode selector, has read access only.
35	The mode selector in module_1 is in PUSH mode.
36	The mode selector in module_1 is in POP mode.
37	The mode selector in module_1 is in CLEAR mode.

TABLE XXXI

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE ADA PROGRAM CALC, CASE 2

Object	Assumptions
A.1	4, 5
A.2	6, 7
A.3	1, 8, 15, 16
A.4	3
A.5	4, 9
A.6	2, 4, 10, 17, 18
A.7	11, 33, 34, 35
A.8	4, 12
A.9	4, 20
A.10	13
A.11	14
B.1	19, 21, 22, 31, 32, 33, 34, 36
B.2	31, 33, 34, 35
B.3	19, 21, 23, 31, 32, 33, 34, 36
B.4	31, 33, 34, 35
B.5	19, 21, 24, 31, 32, 33, 34, 36
B.6	31, 33, 34, 35
B.7	19, 21, 25, 31, 32, 33, 34, 36
B.8	31, 33, 34, 35
B.9	21, 26, 33, 34, 37
B.10	19, 21, 27, 33, 34, 36
B.11	21, 28
B.12	21, 29
B.13	21, 30
CDE.0	34
F.0	16
G.0	18

TABLE XXXII

ASSUMPTIONS FOR THE ADA PROGRAM
CALC, CASE 3

Asmp. Number	Assumption
1	Write access to string variable STR.
2	Write access to float variable NUM_VAL.
3	Write access to boolean variable FIRST.
4	Read access to string variable STR.
5	The first character in string variable STR is not "q" or "Q".
6	Read access to boolean variable FIRST.
7	Boolean variable FIRST is not true.
8	Write access to natural variable LEN.
9	Write access to natural variable CUM_COUNT.
10	Boolean variable END_OF_LINE is not true.
11	Write access to character variable CH.
12	Read access to natural variable CUM_COUNT.
13	Read access to character variable CH.
14	The first character in string variable STR is either a digit, a ".", or a "-" and the value of the natural variable LEN is larger than one.
15	Read access to natural variable LEN.
16	Write access to float variable X.
17	Write access to float variable SIGN.
18	Write access to boolean variable DECIMAL_POINT.
19	Write access to integer variable COUNT.
20	Write access to integer variable EXP.
21	Write access to integer variable EXP_SIGN.
22	Write access to integer variable INDEX.
23	The first character in string variable STR is a "-".
24	Integer variable INDEX is smaller than or equal to natural variable LEN.
25	Read access to integer variable INDEX.
26	The character in character variable CH is a ".".
27	The character in character variable CH is a digit.
28	Read access to float variable X.
29	Boolean variable DECIMAL_POINT is true.
30	Read access to boolean variable DECIMAL_POINT.
31	Read access to integer variable COUNT.
32	The character in character variable CH is either an "e" or an "E".
33	Read access to integer variable JDEX.
34	Write access to integer variable JDEX.
35	Write access to character variable CHR.
36	Read access to character variable CHR.
37	The character in character variable CHR is a digit.

TABLE XXXII (Continued)

Asmp. Number	Assumption
38	Read access to integer variable EXP.
39	The character in character variable CHR is the "-".
40	The character in character variable CHR is not the "-" nor a digit.
41	The character in character variable CHR is not the ".", a digit, "E", nor "e".
42	Read access to float variable SIGN.
43	Read access to integer variable EXP_SIGN.
44	Read access to integer variable EXP.
45	Read access to natural variable TOP.
46	Read access to natural constant LIMIT.
47	Integer variables TOP and LIMIT are equal.
48	Complement of assumption 47.
49	Write access to natural variable TOP.
50	Write access to array of float STACK.
51	Write access to float variable NUM.
52	Read access to float variable NUM_VAL.
53	Read access to float variable NUM.
54	Complement of assumption 14.
55	Write access to string variable STRG.
56	Read access to string variable STRG.
57	The first character in string variable STRG is "+".
58	The first character in string variable STRG is "-".
59	The first character in string variable STRG is "*".
60	The first character in string variable STRG is "/".
61	The first character in string variable STRG is "r" or "R".
62	The first character in string variable STRG is "d" or "D".
63	The first character in string variable STRG is "?".
64	The first character in string variable STRG is "q" or "Q".
65	The first character in string variable STRG is not "+", "-", "*", "/", "r", "R", "d", "D", "?", "q", nor "Q".
66	Read access to float variable Y.
67	Write access to float variable Y.
68	Integer variable TOP equals 0.
69	Complement of assumption 68.
70	Read access to array of float STACK.
71	Exception INVALID_ENTRY has been raised.
72	Exception NUMERIC_ERROR has been raised.

TABLE XXXIII

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE ADA PROGRAM CALC, CASE 3

Object	Assumptions
A.1	4, 5
A.2	6, 7
A.3	1, 8
A.4	3
A.5	4, 14
A.6	2, 4, 15
A.7	51, 52
A.8	4, 54
A.9	4, 55
A.10	71
A.11	72
B.1	28, 53, 56, 57, 66, 67
B.2	51, 66
B.3	28, 53, 56, 58, 66, 67
B.4	51, 66
B.5	28, 53, 56, 59, 66, 67
B.6	51, 66
B.7	28, 53, 56, 60, 66, 67
B.8	51, 66
B.9	56, 61
B.10	28, 53, 56, 62
B.11	56, 63
B.12	56, 64
B.13	56, 65
C.1	45, 46, 47
C.2	45, 46, 48
C.3	45, 49
C.4	45, 50, 53
D.1	45, 68
D.2	45, 69
D.3	45, 51, 70
D.4	45, 49
E.1	49
F.1	10
F.2	11
F.3	9, 12
F.4	1, 12, 13
F.5	8, 12
G.1	4, 23
G.2	17
G.3	22
G.4	15, 24, 25
G.5	4, 11, 25

TABLE XXXIII (Continued)

Object	Assumptions
G.6	13, 18, 26
G.7	13, 16, 27, 28
G.8	29, 30
G.9	19, 31
G.10	13, 32
G.11	15, 25, 33, 34
G.12	4, 33, 35
G.13	20, 36, 37, 38
G.14	21, 36, 39
G.15	36, 40
G.16	15, 22
G.17	13, 41
G.18	22, 25
G.19	2, 28, 31, 42, 43, 44

TABLE XXXIV

ASSUMPTIONS FOR THE ADA PROGRAM
ADDRESS, CASE 1

Asmp. Number	Assumption
1	Write access to pointer PT_LIST of type A_LIST.
2	There exists a record of type LIST with two integer fields called LAST_REC and NEXT_SPACE respectively, a field called SPACE which is an array of type integer, and a field called KEY_LIST which is an array of type KEY.
3	There exists a record of type KEY with a field called NAME of character string type and a field PT_DATA of type positive.
4	Read access to constant MAX_SIZE.
5	Write access to pointer LEN_LIST of type A_LIST.
6	Read access to pointer LEN_LIST of type A_LIST.
7	Read access to pointer PT_LIST of type A_LIST.
8	Read access to boolean variable FIRST.
9	Write access to boolean variable FIRST.
10	Write access to variable OP of type OPERATION.
11	Read access to variable OP of type OPERATION.
12	The value in variable OP is equal to CREATE.
13	The value in variable OP is equal to ADD.
14	The value in variable OP is equal to CHANGE.
15	The value in variable OP is equal to DELETE.
16	The value in variable OP is equal to SEARCH.
17	The value in variable OP is equal to QUIT.
18	Write access to variable DATA of type ADDRESS.
19	Read access to variable DATA of type ADDRESS.
20	Write access to character string variable NAME.
21	Read access to character string variable NAME.
22	Write access to integer variable INDEX.
23	Write access to boolean variable FOUND.
24	Boolean variable FOUND is true.
25	Read access to boolean variable FOUND.
26	There exists a record of type ADDRESS with the following seven character string fields: NAME, STREET, CITY, STATE, ZIP, AREA, and PHONE.
27	Read access to file pointer DATA_ID of type ADDRESS_IO.
28	Read access to field PT_DATA in the record pointed at by the field KEY_LIST in the record pointed at by pointer PT_LIST.
29	Read access to integer variable INDEX.
30	Boolean variable FOUND is not true.

TABLE XXXIV (Continued)

Asmp. Number	Assumption
31	Write access to field LAST_REC in record pointed at pointer LEN_LIST.
32	Read access to field SIZE in record pointed at by pointer PT_LIST.
33	Read access to file pointer INDX_IL of type INDEX_IO.
34	Read access to all fields in record pointed at by pointer LEN_LIST.
35	Read access to all fields in record pointed at by pointer PT_LIST.
36	Read access to character string constant INDX_NAME.
37	Write access to file pointer DATA_ID of type ADDRESS_IO.
38	Write access to file pointer INDX_ID of type INDEX_IO.
39	There exists a procedure GET_NAME with one parameter.
40	The parameter in the procedure GET_NAME has read and write access.
41	Procedure GET_NAME returns a character string obtained from standard input.
42	There exists a procedure START_UP with three parameters.
43	All of the three parameters in procedure START_UP have read and write access.
44	Procedure START_UP returns a pointer to the first record of data, a pointer to the first record of the index, and a boolean indicator that indicates if input has been accepted.
45	There exists a procedure CREATE_LIST with two parameters.
46	Both parameters in procedure CREATE_LIST have read and write access.
47	Procedure CREATE_LIST returns a pointer to the created list of data records and a pointer to the list of index records.
48	There exists a procedure ENTER_DATA with one parameter.
49	The parameter in procedure ENTER_DATA has write access only.
50	Procedure ENTER_DATA returns a pointer to a newly created data record.
51	There exists a procedure DISPLAY with one parameter.

TABLE XXXIV (Continued)

Asmp. Number	Assumption
52	The parameter in procedure DISPLAY has read access only.
53	There exists a procedure ALTER_DATA with one parameter.
54	The parameter in procedure ALTER_DATA has read and write access.
55	Procedure ALTER_DATA returns a pointer to the record for which the data has been altered.
56	There exists a procedure ALTER_FIELD with one parameter.
57	The parameter in procedure ALTER_FIELD has read and write access.
58	Procedure ALTER_FIELD returns a new character string accepted from standard input or the same character string received as input if no changes were desired.
59	There exists a procedure SELECT_ALTERNATIVE with two parameters.
60	One parameter in procedure SELECT_ALTERNATIVE has both read and write access, the other has only write access.
61	Procedure SELECT_ALTERNATIVE returns the type of operation desired to be performed on the database.
62	There exists a procedure INSERT with two parameters.
63	One parameter in procedure INSERT has both read and write access, the other has only read access.
64	Procedure INSERT returns a pointer to the first record after the insertion has taken place.
65	There exists a procedure SEARCH with five parameters.
66	Two parameters in procedure SEARCH have read access only, the other three have both read and write access.
67	Procedure SEARCH searches the index for a name and if found, returns the address of the record and indicates that the record was found.
68	There exists a procedure DELETE with two parameters.
69	One parameter in procedure DELETE has both read and write access, the other has only read access.
70	Procedure DELETE returns a pointer to the first record after the deletion has taken place.
71	There exists a procedure GET_STRING with two parameters.
72	Both parameters in procedure GET_STRING have write access only.

TABLE XXXIV (Continued)

Asmp. Number	Assumption
73	Procedure GET_STRING returns a character string obtained from standard input and the number of characters in the returned string.

TABLE XXXV

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE ADA PROGRAM ADDRESS, CASE 1

Object	Assumptions
A.1	1, 2, 3, 4
A.2	2, 3, 4, 5
A.3	1, 5, 6, 7, 8, 9, 42, 43, 44
A.4	8, 9, 10, 59, 60, 61
A.5	1, 5, 6, 7, 11, 12, 45, 46, 47
A.6	11, 13, 18, 48, 49, 50
A.7	1, 7, 19, 62, 63, 64
A.8	11, 14, 20, 21, 39, 40, 41
A.9	7, 18, 21, 22, 23, 65, 66, 67
A.10	24, 25
A.11	18, 19, 53, 54, 55
A.12	2, 3, 19, 27, 28, 29
A.13	25, 30
A.14	11, 15, 20, 21, 39, 40, 41
A.15	1, 7, 21, 68, 69, 70
A.16	11, 16, 20, 21, 39, 40, 41
A.17	7, 18, 21, 22, 23, 65, 66, 67
A.18	24, 25
A.19	19, 51, 52
A.20	25, 30
A.21	11, 17
A.22	11, 17
A.23	2, 31, 32
A.24	2, 33, 34
A.25	2, 33, 35
A.26	33
A.27	27
B.0	40, 41, 71, 72, 73
C.0	2, 27, 33, 36, 37, 38, 43, 44, 45, 46, 47, 71, 72
D.0	2, 3, 4, 26, 27, 36, 37, 38, 46, 47, 48, 49, 50
E.0	26, 49, 50, 51, 52, 53, 54, 55
F.0	26, 52
G.0	26, 51, 52, 54, 55, 56, 57, 58, 71, 72, 73
H.0	57, 58, 71, 72, 73
I.0	60, 61, 71, 72, 73
J.0	2, 3, 4, 26, 27, 51, 52, 63, 64, 65, 66, 67, 71, 72, 73
K.0	2, 66, 67
L.0	2, 3, 4, 51, 52, 65, 66, 67, 69, 70, 71, 72, 73
M.0	72, 73

TABLE XXXVI

ASSUMPTIONS FOR THE ADA PROGRAM
ADDRESS, CASE 2

Asmp. Number	Assumption
1	Write access to pointer PT_LIST of type A_LIST.
2	There exists a record of type LIST with two integer fields called LAST_REC and NEXT_SPACE respectively, a field called SPACE which is an array of type integer, and a field called KEY_LIST which is an array of type KEY.
3	There exists a record of type KEY with a field called NAME of character string type and a field PT_DATA of type positive.
4	Read access to constant MAX_SIZE.
5	Write access to pointer LEN_LIST of type A_LIST.
6	Read access to pointer LEN_LIST of type A_LIST.
7	Read access to pointer PT_LIST of type A_LIST.
8	Read access to boolean variable FIRST.
9	Write access to boolean variable FIRST.
10	Write access to variable OP of type OPERATION.
11	Read access to variable OP of type OPERATION.
12	The value in variable OP is equal to CREATE.
13	The value in variable OP is equal to ADD.
14	The value in variable OP is equal to CHANGE.
15	The value in variable OP is equal to DELETE.
16	The value in variable OP is equal to SEARCH.
17	The value in variable OP is equal to QUIT.
18	Write access to variable DATA of type ADDRESS.
19	Read access to variable DATA of type ADDRESS.
20	Write access to character string variable NAME.
21	Read access to character string variable NAME.
22	Write access to integer variable INDEX.
23	Write access to boolean variable FOUND.
24	Boolean variable FOUND is true.
25	Read access to boolean variable FOUND.
26	There exists a record of type ADDRESS with the following seven character string fields: NAME, STREET, CITY, STATE, ZIP, AREA, and PHONE.
27	Read access to file pointer DATA_ID of type ADDRESS_IO.
28	Read access to field PT_DATA in the record pointed at by the field KEY_LIST in the record pointed at by pointer PT_LIST.
29	Read access to integer variable INDEX.
30	Boolean variable FOUND is not true.

TABLE XXXVI (Continued)

Asmp. Number	Assumption
31	Write access to field LAST_REC in record pointed at pointer LEN_LIST.
32	Read access to field SIZE in record pointed at by pointer PT_LIST.
33	Read access to file pointer INDX_ID of type INDEX_IO.
34	Read access to all fields in record pointed at by pointer LEN_LIST.
35	Read access to all fields in record pointed at by pointer PT_LIST.
36	Read access to character string constant INDX_NAME.
37	Write access to file pointer DATA_ID of type ADDRESS_IO.
38	Write access to file pointer INDX_ID of type INDEX_IO.
39	There exists a procedure GET_NAME with one parameter.
40	The parameter in the procedure GET_NAME has read and write access.
41	Procedure GET_NAME returns a character string obtained from standard input.
42	There exists a procedure START_UP with three parameters.
43	All of the three parameters in procedure START_UP have read and write access.
44	Procedure START_UP returns a pointer to the first record of data, a pointer to the first record of the index, and a boolean indicator that indicates if input has been accepted.
45	There exists a procedure CREATE_LIST with two parameters.
46	Both parameters in procedure CREATE_LIST have read and write access.
47	Procedure CREATE_LIST returns a pointer to the created list of data records and a pointer to the list of index records.
48	There exists a procedure ENTER_DATA with one parameter.
49	The parameter in procedure ENTER_DATA has write access only.
50	Procedure ENTER_DATA returns a pointer to a newly created data record.
51	There exists a procedure DISPLAY with one parameter.

TABLE XXXVI (Continued)

Asmp. Number	Assumption
52	The parameter in procedure DISPLAY has read access only.
53	There exists a module called MODULE_1 with one parameter.
54	The parameter in module MODULE_1 has read and write access.
55	Module MODULE_1 returns a pointer to the record for which the data has been altered.
56	There exists a procedure SELECT_ALTERNATIVE with two parameters.
57	One parameter in procedure SELECT_ALTERNATIVE has both read and write access, the other has only write access.
58	Procedure SELECT_ALTERNATIVE returns the type of operation desired to be performed on the database.
59	There exists a procedure SEARCH with five parameters.
60	Two parameters in procedure SEARCH have read access only, the other three have both read and write access.
61	Procedure SEARCH searches the index for a name and if found, returns the address of the record and indicates that the record was found.
62	There exists a procedure GET_STRING with two parameters.
63	Both parameters in procedure GET_STRING have write access only.
64	Procedure GET_STRING returns a character string obtained from standard input and the number of characters in the returned string.
65	Write access to integer variable COUNT.
66	Write access to integer variable I.
67	Read access to integer variable I.
68	Write access to character string RESPONSE.
69	Read access to character string RESPONSE.
70	Read access to field NAME in record pointed at by pointer DATA of type ADDRESS.
71	Read access to field NAME in record pointed at by pointer DATA.
72	The first character in character string RESPONSE is either a "y" or a "Y".
73	Write access to pointer TEMP_DATA of type ADDRESS.
74	Read access to pointer TEMP_DATA of type ADDRESS.
75	The first character in character string RESPONSE is either a "o" or a "O".

TABLE XXXVI (Continued)

Asmp. Number	Assumption
76	Complement of assumption number 111.
77	Write access to natural variable REC_NUM.
78	Write access to pointer NEW_LIST of type A_LIST.
79	Read access to field KEY_LIST in record pointed at by pointer PT_LIST.
80	Write access to field KEY_LIST in record pointed at by pointer NEW_LIST.
81	Write access to field NAME pointed at by the field KEY_LIST in the record pointed at by pointer NEW_LIST.
82	Field NEXT_SPACE in record pointed at by pointer PT_LIST is zero.
83	Read access to field NEXT_SPACE in record pointed at by pointer NEW_LIST.
84	Write access to field LAST_REC in record pointed at by pointer NEW_LIST.
85	Read access to field LAST_REC in record pointed at by pointer PT_LIST.
86	Read access to field LAST_REC in record pointed at by pointer NEW_LIST.
87	Write access to field NEXT_SPACE in record pointed at by pointer NEW_LIST.
88	Complement of assumption number 118.
89	Read access to field SPACE in record pointed at by pointer PT_LIST.
90	Read access to field NEXT_SPACE in record pointed at by pointer PT_LIST.
91	Write access to field SPACE in record pointed at by pointer NEW_LIST.
92	Read access to natural variable REC_NUM.
93	Write access to field PT_DATA in record pointed at by fld KEY_LIST in rec. pointed at by ptr. NEW_LIST.
94	Read access to field SIZE in record pointed at by pointer NEW_LIST.
95	Read access to pointer NEW_LIST of type A_LIST.
96	Read access to natural variable REC_NUM.
97	Read access to character string DEL_NAME.
98	Write access to field KEY_LIST in record pointed at by pointer PT_LIST.
99	Read access to field SPACE in record pointed at by pointer NEW_LIST.
100	The last entry in the arrayed field SPACE in record pointed at by ptr. NEW_LIST is larger than the field NEXT_SPACE in record pointed at by pointer NEW_LIST.

TABLE XXXVII

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE ADA PROGRAM ADDRESS, CASE 2

Object	Assumptions
A.1	1, 2, 3, 4
A.2	2, 3, 4, 5
A.3	1, 5, 6, 7, 8, 9, 42, 43, 44
A.4	8, 9, 10, 56, 57, 58
A.5	1, 5, 6, 7, 11, 12, 45, 46, 47
A.6	11, 13, 18, 48, 49, 50
A.7	1, 7, 19
A.8	11, 14, 20, 21, 39, 40, 41
A.9	7, 18, 21, 22, 23, 59, 60, 61
A.10	24, 25
A.11	18, 19, 53, 54, 55
A.12	2, 3, 19, 27, 28, 29
A.13	25, 30
A.14	11, 15, 20, 21, 39, 40, 41
A.15	1, 7, 21
A.16	11, 16, 20, 21, 39, 40, 41
A.17	7, 18, 21, 22, 23, 59, 60, 61
A.18	24, 25
A.19	19, 51, 52
A.20	25, 30
A.21	11, 17
A.22	11, 17
A.23	2, 31, 32
A.24	2, 33, 34
A.25	2, 33, 35
A.26	33
A.27	27
B.0	40, 41, 62, 63, 64
C.0	2, 27, 33, 36, 37, 38, 43, 44, 45, 46, 47, 62, 63
D.0	2, 3, 4, 26, 27, 36, 37, 38, 46, 47, 48, 49, 50
E.0	26, 49, 50, 51, 52, 53, 54, 55
F.0	26, 52
GH.0	26, 51, 52, 54, 55, 62, 63, 64
I.0	57, 58, 62, 63, 64
J.1	7, 19, 22, 23, 59, 60, 61, 71, 73
J.2	24, 25
J.3	51, 52, 82
J.4	62, 63, 64, 65, 68
J.5	69, 75
J.6	69, 76
J.7	2, 3, 28, 29, 77
J.8	25, 30

TABLE XXXVII (Continued)

Object	Assumptions
J.9	2, 3, 4, 32, 78
J.10	29, 66, 67
J.11	2, 67, 79, 80
J.12	22, 29
J.13	2, 3, 26, 29, 70, 81
J.14	2, 82, 83
J.15	2, 84, 85
J.16	2, 77, 86
J.17	2, 87
J.18	2, 83, 88
J.19	2, 84, 85
J.20	2, 77, 89, 90
J.21	2, 87, 90
J.22	2, 89, 91
J.23	2, 3, 29, 92, 93
J.24	2, 29, 66, 67, 94
J.25	2, 67, 79, 80
J.26	1, 95
J.27	19, 27, 96
K.0	2, 60, 61
L.1	7, 22, 23, 59, 60, 61, 73, 97
L.2	25, 30
L.3	24, 25
L.4	51, 52
L.5	62, 63, 64, 65, 68
L.6	69, 72
L.7	2, 3, 28, 29, 77
L.8	29, 32, 66, 67
L.9	67, 79, 98
L.10	2, 3, 4, 78, 80, 84, 87, 91
L.11	2, 83, 99, 100
L.12	2, 83, 87
L.13	2, 83, 91, 92
L.14	1, 95
M.0	72, 73

TABLE XXXVIII

ASSUMPTIONS FOR THE ADA PROGRAM
ADDRESS, CASE 3

Asmp. Number	Assumption
1	Write access to pointer PT_LIST of type A_LIST.
2	There exists a record of type LIST with two integer fields called LAST_REC and NEXT_SPACE respectively, a field called SPACE which is an array of type integer, and a field called KEY_LIST which is an array of type KEY.
3	There exists a record of type KEY with a field called NAME of character string type and a field PT_DATA of type positive.
4	Read access to constant MAX_SIZE.
5	Write access to pointer LEN_LIST of type A_LIST.
6	Read access to pointer LEN_LIST of type A_LIST.
7	Read access to pointer PT_LIST of type A_LIST.
8	Read access to boolean variable FIRST.
9	Write access to boolean variable FIRST.
10	Write access to variable OP of type OPERATION.
11	Read access to variable MODE of type OPERATION.
12	Read access to variable OP of type OPERATION.
13	The value in variable OP is equal to CREATE.
14	The value in variable OP is equal to ADD.
15	The value in variable OP is equal to CHANGE.
16	The value in variable OP is equal to DELETE.
17	The value in variable OP is equal to SEARCH.
18	The value in variable OP is equal to QUIT.
19	Write access to variable DATA of type ADDRESS.
20	Read access to variable DATA of type ADDRESS.
21	Write access to character string variable NAME.
22	Read access to character string variable NAME.
23	Write access to character string variable SEEK_NAME.
24	Write access to integer variable INDEX.
25	Write access to boolean variable FOUND.
26	Boolean variable FOUND is true.
27	Read access to boolean variable FOUND.
28	There exists a record of type ADDRESS with the following seven character string fields: NAME, STREET, CITY, STATE, ZIP, AREA, and PHONE.
29	Read access to file pointer DATA_ID of type ADDRESS_IO.
30	Read access to field PT_DATA in the record pointed at by the field KEY_LIST in the record pointed at by pointer PT_LIST.
31	Read access to integer variable INDEX.

TABLE XXXVIII (Continued)

Asmp. Number	Assumption
32	Boolean variable FOUND is not true.
33	Write access to field LAST_REC in record pointed at pointer LEN_LIST.
34	Read access to field SIZE in record pointed at by pointer PT_LIST.
35	Read access to file pointer INDX_ID of type INDEX_IO.
36	Read access to all fields in record pointed at by pointer LEN_LIST.
37	Read access to all fields in record pointed at by pointer PT_LIST.
38	Write access to integer variable COUNT.
39	Read access to integer variable LEN.
40	Read access to character string variable STR.
41	Write access to integer variable I.
42	Read access to integer variable I.
43	Read access to integer variable COUNT.
44	Read access to character string constant INDX_NAME.
45	Write access to all fields in record pointed at by pointer LEN_LIST.
46	Read access to field LAST_REC in record pointed at by pointer LEN_LIST.
47	Write access to all fields in record pointed at by pointer PT_LIST.
48	Exception INDEX_IO.NAME_ERROR has been raised.
49	Write access to character string RESPONSE.
50	Write access to natural variable LEN.
51	The first character in character string RESPONSE is either a "q" or a "Q".
52	Read access to character string RESPONSE.
53	Complement of assumption number 51.
54	File pointer DATA_ID of type ADDRESS_IO is null, i.e., file is not open.
55	Write access to file pointer DATA_ID of type ADDRESS_IO.
56	Write access to file pointer INDX_ID of type INDEX_IO.
57	Read access to character string constant DATA_NAME.
58	Exception ADDRESS_IO.NAME_ERROR has been raised.
59	Boolean variable FIRST is true.
60	Boolean variable FIRST is not true.
61	Exception ADDRESS_IO.STATUS_ERROR has been raised.
62	Exception INDEX_IO.STATUS_ERROR has been raised.
63	Read access to field NAME in record pointed at by pointer DATA of type ADDRESS.

TABLE XXXVIII (Continued)

Asmp. Number	Assumption
64	Write access to field NAME in record pointed at by pointer INIT_KEY of type KEY.
65	Write access to field PT_DATA in record pointed at by pointer INIT_KEY of type KEY.
66	Read access to pointer INIT_KEY of type KEY.
67	Write access to field NEXT_SPACE in record pointed at by pointer LEN_LIST.
68	Write access to field SPACE in record pointed at by pointer LEN_LIST.
69	Write access to field NAME in record pointed at by pointer NEW_ADDRESS.
70	Write access to field STREET in record pointed at by pointer NEW_ADDRESS.
71	Write access to field CITY in record pointed at by pointer NEW_ADDRESS.
72	Write access to field STATE in record pointed at by pointer NEW_ADDRESS.
73	Write access to field ZIP in record pointed at by pointer NEW_ADDRESS.
74	Write access to field AREA in record pointed at by pointer NEW_ADDRESS.
75	Write access to field PHONE in record pointed at by pointer NEW_ADDRESS.
76	Read access to pointer NEW_ADDRESS of type ADDRESS.
77	Write access to pointer NEW_ADDRESS of type ADDRESS.
78	Read access to field NAME in record pointed at by pointer DATA.
79	Read access to field STREET in record pointed at by pointer DATA.
80	Read access to field CITY in record pointed at by pointer DATA.
81	Read access to field STATE in record pointed at by pointer DATA.
82	Read access to field ZIP in record pointed at by pointer DATA.
83	Read access to field AREA in record pointed at by pointer DATA.
84	Read access to field PHONE in record pointed at by pointer DATA.
85	Write access to field NAME in record pointed at by pointer DATA.
86	Write access to field STREET in record pointed at by pointer DATA.
87	Write access to field CITY in record pointed at by pointer DATA.

TABLE XXXVIII (Continued)

Asmp. Number	Assumption
88	Write access to field STATE in record pointed at by pointer DATA.
89	Write access to field ZIP in record pointed at by pointer DATA.
90	Write access to field AREA in record pointed at by pointer DATA.
91	Write access to field PHONE in record pointed at by pointer DATA.
92	Write access to natural variable NUM_CHAR.
93	The first character in character string RESPONSE is either a "y" or a "Y".
94	Read access to character string STRG.
95	Write access to character string STRG.
96	Read access to character string REPLY.
97	Write access to character string REPLY.
98	Write access to natural variable IN_CHAR.
99	Natural variable IN_CHAR is larger than zero.
100	Read access to natural variable IN_CHAR.
101	The first character in character string RESPONSE is either a "i" or a "I".
102	The first character in character string RESPONSE is either a "c" or a "C".
103	The first character in character string RESPONSE is either a "a" or a "A".
104	The first character in character string RESPONSE is either a "d" or a "D".
105	The first character in character string RESPONSE is either a "s" or a "S".
106	Write access to variable MODE of type OPERATION.
107	The first character in character string RESPONSE is not an "i", "I", "c", "C", "a", "A", "d", "D", "s", nor "S".
108	Write access to pointer TEMP_DATA of type ADDRESS.
109	Write access to character string SEEK_NAME.
110	Read access to pointer TEMP_DATA of type ADDRESS.
111	The first character in character string RESPONSE is either a "o" or a "O".
112	Complement of assumption number 111.
113	Write access to natural variable REC_NUM.
114	Write access to pointer NEW_LIST of type A_LIST.
115	Read access to field KEY_LIST in record pointed at by pointer PT_LIST.
116	Write access to field KEY_LIST in record pointed at by pointer NEW_LIST.

TABLE XXXVIII (Continued)

Asmp. Number	Assumption
117	Write access to field NAME pointed at by the field KEY_LIST in the record pointed at by pointer NEW_LIST.
118	Field NEXT_SPACE in record pointed at by pointer PT_LIST is zero.
119	Read access to field NEXT_SPACE in record pointed at by pointer NEW_LIST.
120	Write access to field LAST_REC in record pointed at by pointer NEW_LIST.
121	Read access to field LAST_REC in record pointed at by pointer PT_LIST.
122	Read access to field LAST_REC in record pointed at by pointer NEW_LIST.
123	Write access to field NEXT_SPACE in record pointed at by pointer NEW_LIST.
124	Complement of assumption number 118.
125	Read access to field SPACE in record pointed at by pointer PT_LIST.
126	Read access to field NEXT_SPACE in record pointed at by pointer PT_LIST.
127	Write access to field SPACE in record pointed at by pointer NEW_LIST.
128	Read access to natural variable REC_NUM.
129	Write access to field PT_DATA in record pointed at by field KEY_LIST in record pointed at by pointer NEW_LIST.
130	Read access to field SIZE in record pointed at by pointer NEW_LIST.
131	Read access to pointer NEW_LIST of type A_LIST.
132	Read access to natural variable REC_NUM.
133	Read access to integer variable NEXT.
134	Read access to integer variable LAST.
135	Read access to integer variable LLAST.
136	Variable NEXT is not equal to variable LAST.
137	Variable NEXT is not equal to variable LLAST.
138	Write access to character string THIS_NAME.
139	Read access to field NAME in record pointed at by field KEY_LIST in record pointed at by pointer PT_LIST.
140	Read access to character string SEEK_NAME.
141	Read access to character string THIS_NAME.
142	Character strings SEEK_NAME and THIS_NAME contain the same string.

TABLE XXXVIII (Continued)

Asmp. Number	Assumption
143	The character string in variable SEEK_NAME succeeds alphabetically the character string in variable THIS_NAME.
144	Write access to integer variable LOW.
145	Write access to integer variable LLAST.
146	Write access to integer variable LAST.
147	Write access to integer variable NEXT.
148	Read access to integer variable HIGH.
149	The character string in variable THIS_NAME succeeds alphabetically the character string in variable SEEK_NAME.
150	Write access to integer variable HIGH.
151	Read access to integer variable LOW.
152	Read access to pointer BLANKS of type ADDRESS.
153	Read access to character string DEL_NAME.
154	Write access to field KEY_LIST in record pointed at by pointer PT_LIST.
155	Read access to field SPACE in record pointed at by pointer NEW_LIST.
156	The last entry in the arrayed field SPACE in record pointed at by pointer NEW_LIST is larger than the field NEXT_SPACE in record pointed at by pointer NEW_LIST.
157	END_OF_LINE boolean indicator is not true.
158	Write access to character variable CH.
159	Read access to character variable CH.
160	Write access to natural variable CUM_COUNT.
161	Read access to natural variable CUM_COUNT.
162	Write access to character string STR.
163	Write access to natural variable LEN.

TABLE XXXIX

LIST OF ASSUMPTION NUMBERS FOR OBJECTS
IN THE ADA PROGRAM ADDRESS, CASE 3

Object	Assumptions
A.1	1, 2, 3, 4
A.2	2, 3, 4, 5
A.3	1, 5, 6, 7, 8, 9
A.4	8, 9, 10, 11
A.5	1, 5, 6, 7, 12, 13
A.6	12, 14, 19
A.7	1, 7, 20
A.8	12, 15, 21, 22
A.9	7, 19, 22, 23, 24, 25
A.10	26, 27
A.11	19, 20
A.12	2, 3, 20, 29, 30, 31
A.13	27, 32
A.14	12, 16, 21, 22
A.15	1, 7, 22
A.16	12, 17, 21, 22
A.17	7, 19, 22, 23, 24, 25
A.18	26, 27
A.19	20
A.20	27, 32
A.21	12, 18
A.22	12, 18
A.23	2, 33, 34
A.24	2, 35, 36
A.25	2, 35, 37
A.26	35
A.27	29
B.1	21, 38, 39, 40
B.2	22, 41, 42, 43
B.3	21, 42
C.1	44, 56
C.2	2, 35, 45
C.3	1, 2, 46
C.4	2, 35, 47
C.5	48
C.6	39, 40, 49, 50
C.7	9
C.8	51, 52
C.9	52, 53
C.10	1, 5, 6, 7
C.11	29, 54
C.12	55, 57
C.13	58

TABLE XXXIX (Continued)

Object	Assumptions
C.14	8, 59
C.15	9
C.16	8, 60
C.17	39, 40, 49, 50
C.18	51, 52
C.19	52, 53
C.20	56
C.21	1, 5, 6, 7
D.1	55, 57
D.2	61
D.3	55
D.4	55, 57
D.5	44, 56
D.6	62
D.7	56
D.8	44, 56
D.9	19
D.10	20, 29
D.11	28, 63, 64
D.12	28, 65
D.13	1, 2, 3, 4, 47, 66
D.14	2, 33
D.15	2, 67
D.16	2, 68
E.1	28, 38, 39, 40, 69
E.2	28, 38, 39, 40, 70
E.3	28, 38, 39, 40, 71
E.4	28, 38, 39, 40, 72
E.5	28, 38, 39, 40, 73
E.6	28, 38, 39, 40, 74
E.7	28, 38, 39, 40, 75
E.8	76
E.9	38, 39, 40, 49
E.10	52, 93
E.11	76, 77
E.12	19, 76
F.1	28, 78
F.2	28, 79
F.3	28, 80, 81
F.4	28, 82
F.5	28, 83, 84
G.1	28, 85
G.2	28, 86
G.3	28, 87
G.4	28, 88
G.5	28, 89

TABLE XXXIX (Continued)

Object	Assumptions
G.6	28, 90
G.7	28, 91
G.8	20
G.9	49
G.10	39, 40, 49, 92
G.11	52, 93
H.1	94
H.2	39, 40, 97, 98
H.3	99, 100
H.4	95, 96
I.1	8, 60
I.2	39, 40, 43, 49
I.3	9
I.4	52, 101, 106
I.5	38, 39, 40, 49
I.6	52, 93
I.7	52, 102, 106
I.8	52, 103, 106
I.9	52, 104, 106
I.10	52, 105, 106
I.11	51, 52, 106
I.12	38, 39, 40, 49
I.13	52, 93
I.14	52, 107
J.1	7, 20, 24, 25, 78, 108, 109
J.2	26, 27
J.3	110
J.4	38, 39, 40, 49
J.5	52, 111
J.6	52, 112
J.7	2, 3, 30, 31, 113
J.8	27, 32
J.9	2, 3, 4, 34, 114
J.10	31, 41, 42
J.11	2, 42, 115, 116
J.12	24, 31
J.13	2, 3, 28, 31, 63, 117
J.14	2, 118, 119
J.15	2, 120, 121
J.16	2, 113, 122
J.17	2, 123
J.18	2, 119, 124
J.19	2, 120, 121
J.20	2, 113, 125, 126
J.21	2, 113, 126
J.22	2, 125, 127

TABLE XXXIX (Continued)

Object	Assumptions
J.23	2, 3, 31, 128, 129
J.24	2, 31, 41, 42, 130
J.25	2, 42, 115, 116
J.26	1, 131
J.27	20, 29, 132
K.1	133, 134, 135, 136, 137
K.2	2, 133, 138, 139
K.3	140, 141, 142
K.4	25
K.5	24, 133
K.6	2, 20, 29, 30, 133
K.7	140, 141, 143
K.8	133, 144
K.9	134, 145
K.10	133, 146
K.11	133, 147, 148
K.12	140, 141, 149
K.13	133, 150
K.14	134, 145
K.15	133, 146
K.16	133, 147, 151
K.17	25
K.18	19, 152
K.19	140, 141, 143
K.20	24, 134
K.21	140, 141, 149
K.22	24, 134
L.1	7, 24, 25, 108, 109, 153
L.2	27, 32
L.3	26, 27
L.4	110
L.5	38, 39, 40, 49
L.6	52, 93
L.7	2, 3, 30, 31, 113
L.8	31, 34, 41, 42
L.9	2, 42, 115, 154
L.10	2, 3, 4, 37, 114, 116, 120, 123, 127
L.11	2, 119, 155, 156
L.12	2, 119, 123
L.13	2, 119, 127, 128
L.14	1, 131
M.1	157
M.2	158
M.3	160, 161
M.4	159, 161, 162
M.5	161, 163

APPENDIX E

EXECUTION FLOW CHARTS FOR PROGRAMS INCLUDED IN THE STUDY

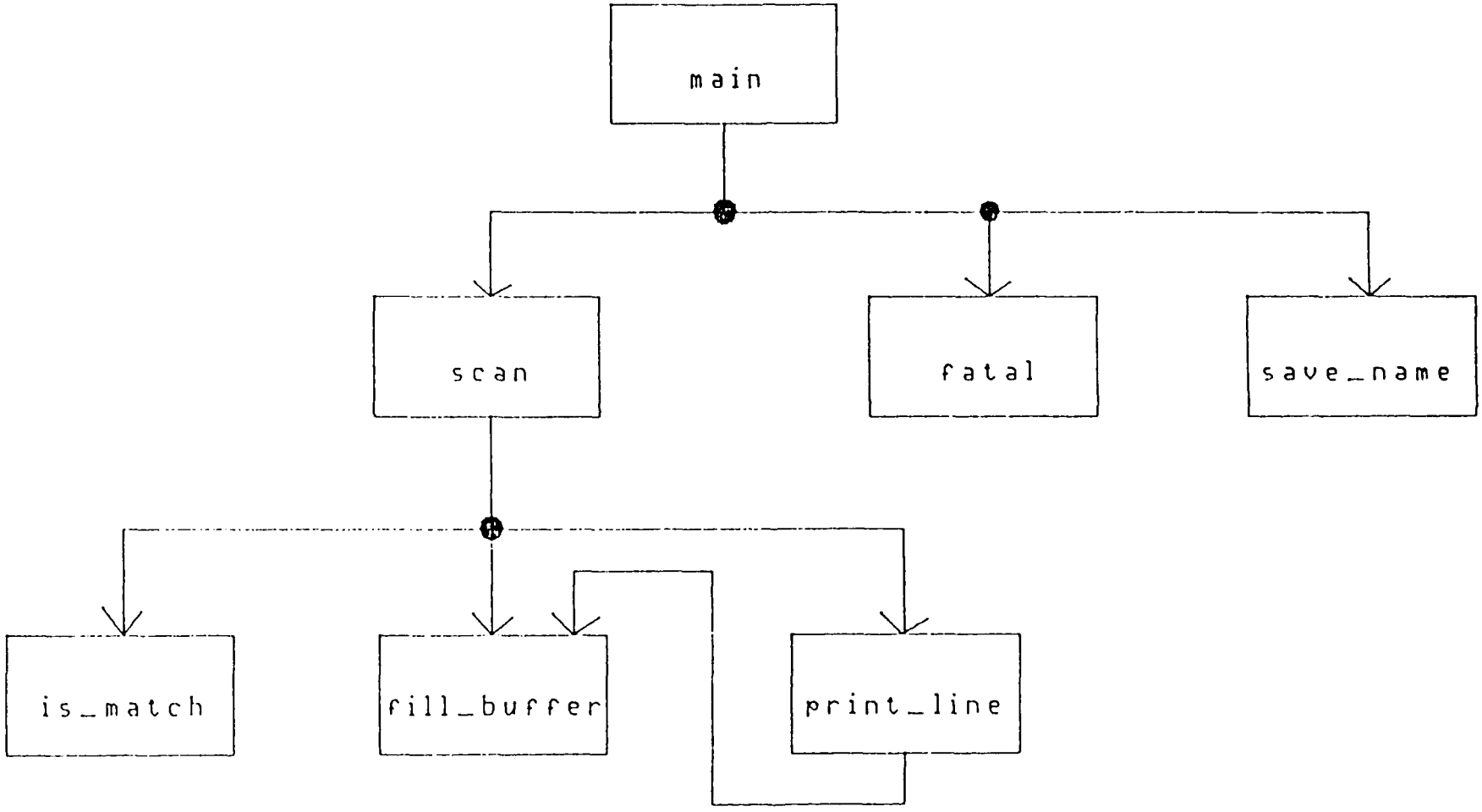


Figure 2. Execution Flow Chart for the C Program fastfind.

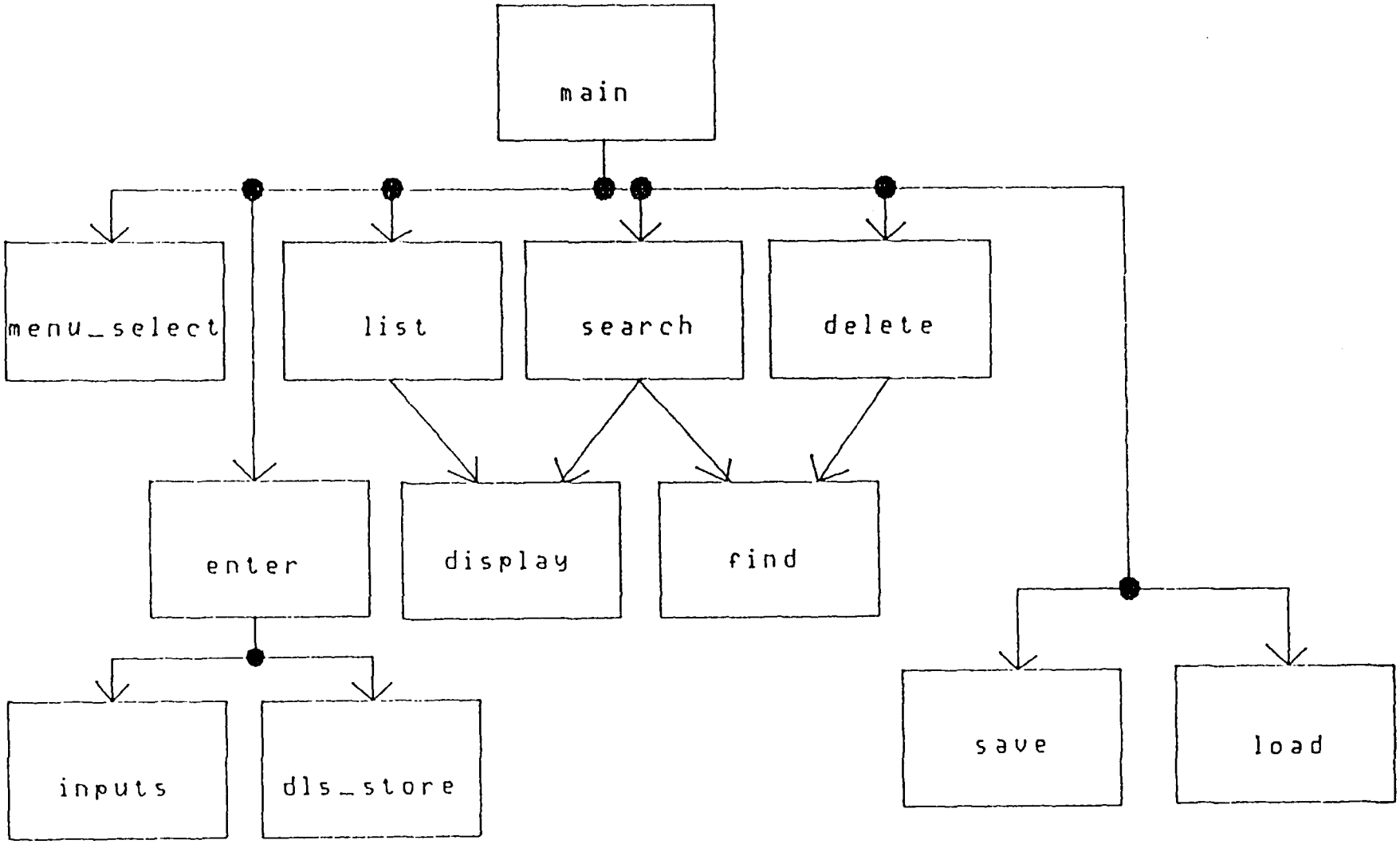


Figure 3. Execution Flow Chart for the C Program mail.

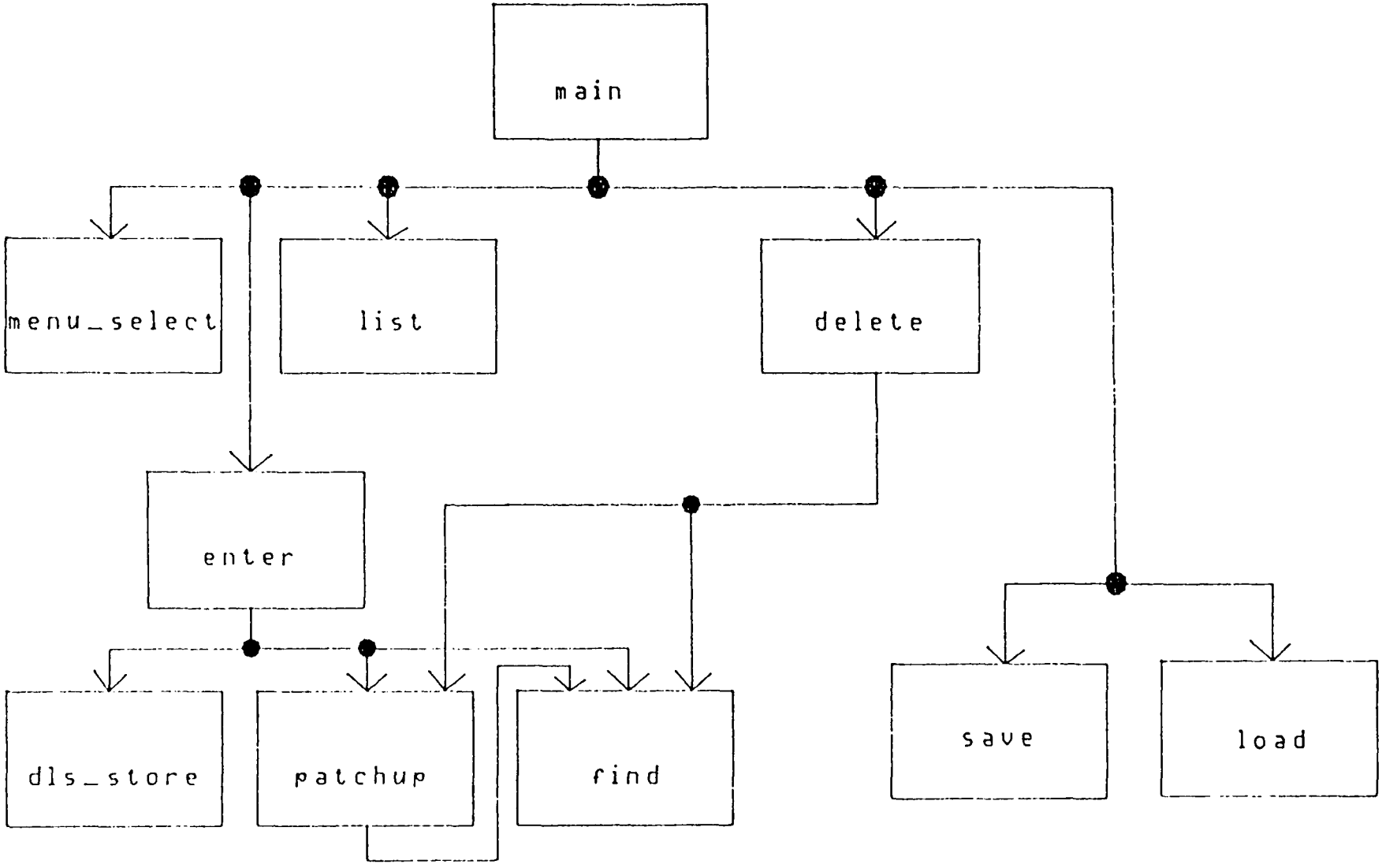


Figure 4. Execution Flow Chart for the C Program editor.

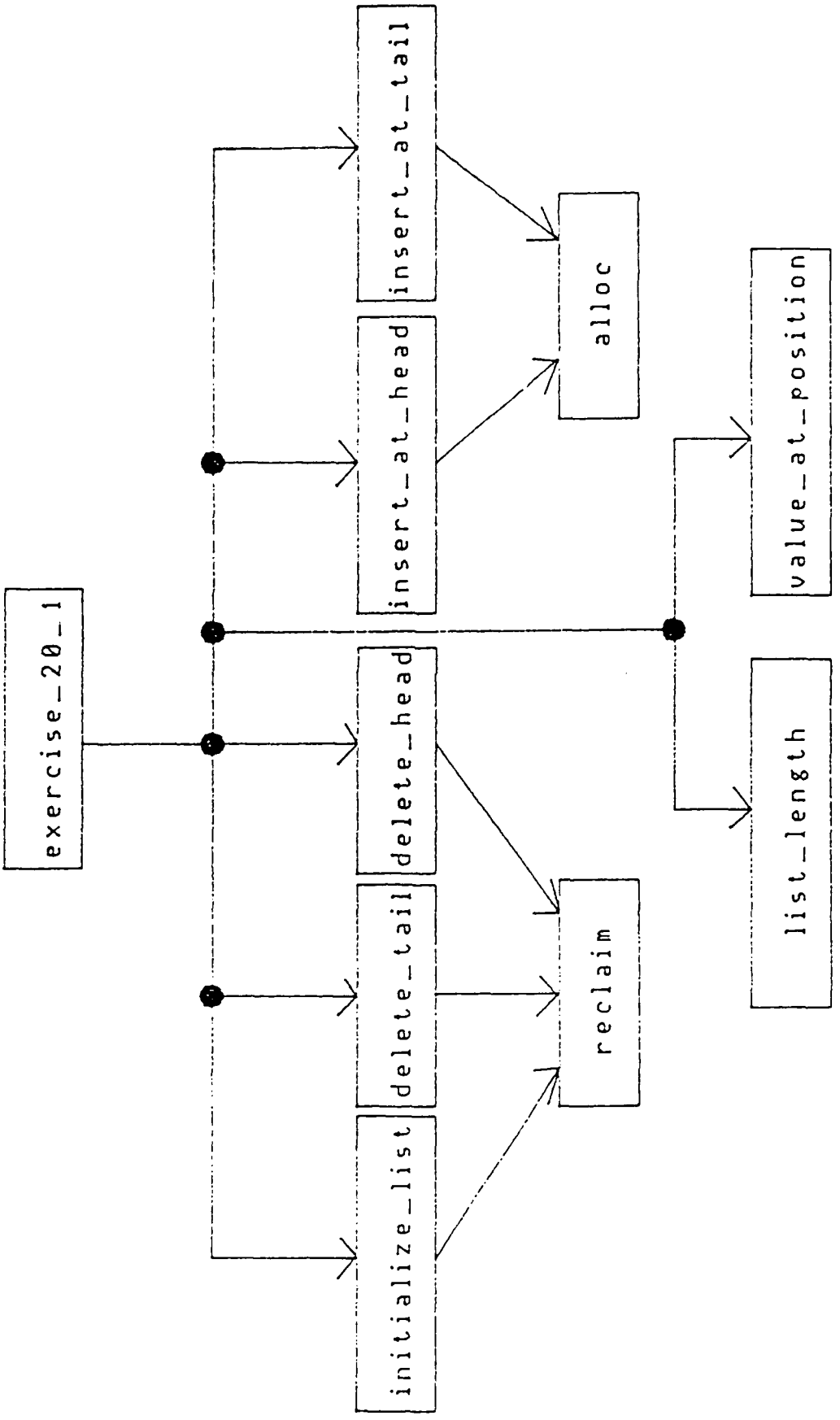


Figure 5. Execution Flow Chart for the Ada Program intlist.

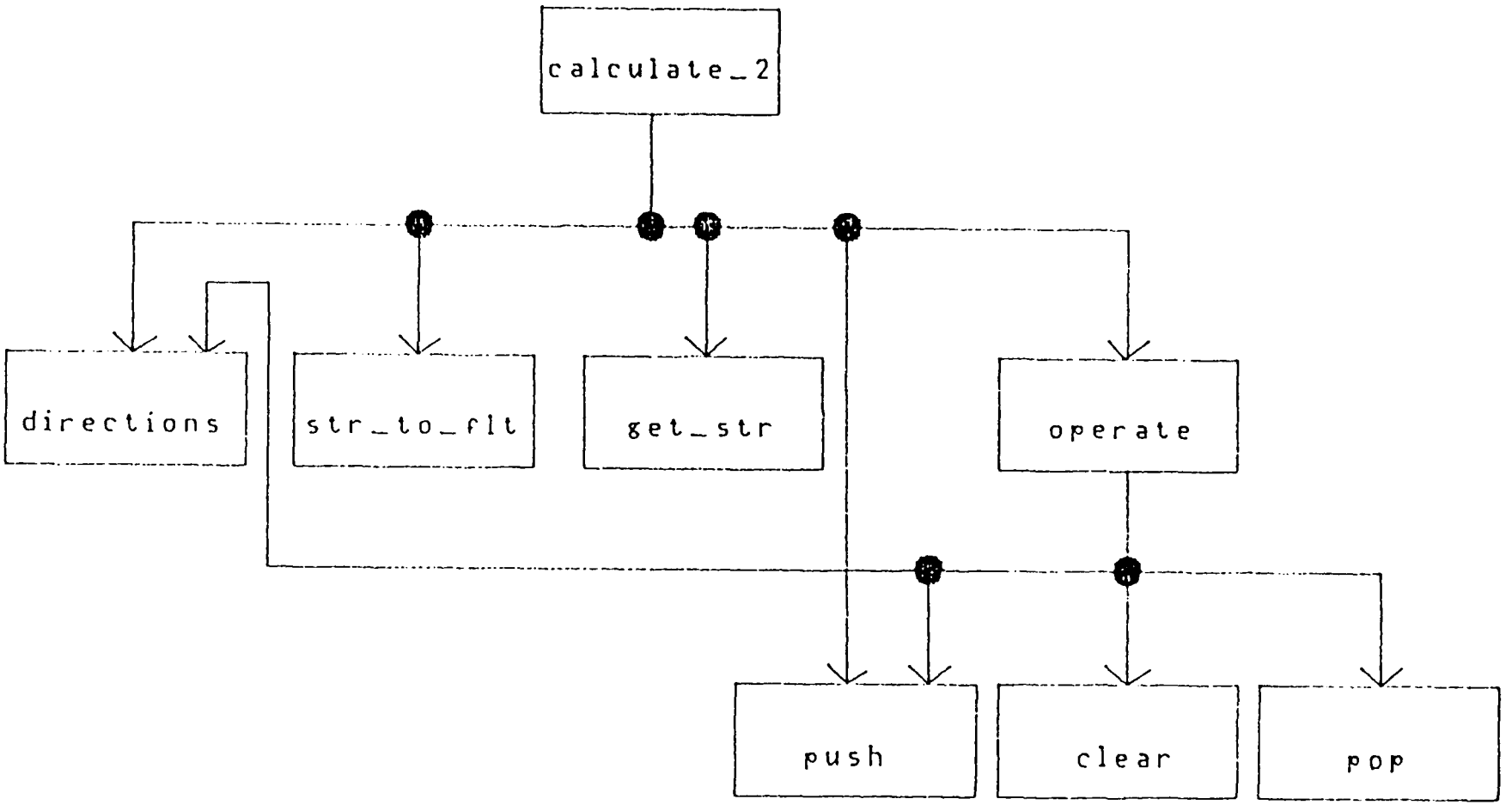


Figure 6. Execution Flow Chart for the Ada Program calc.

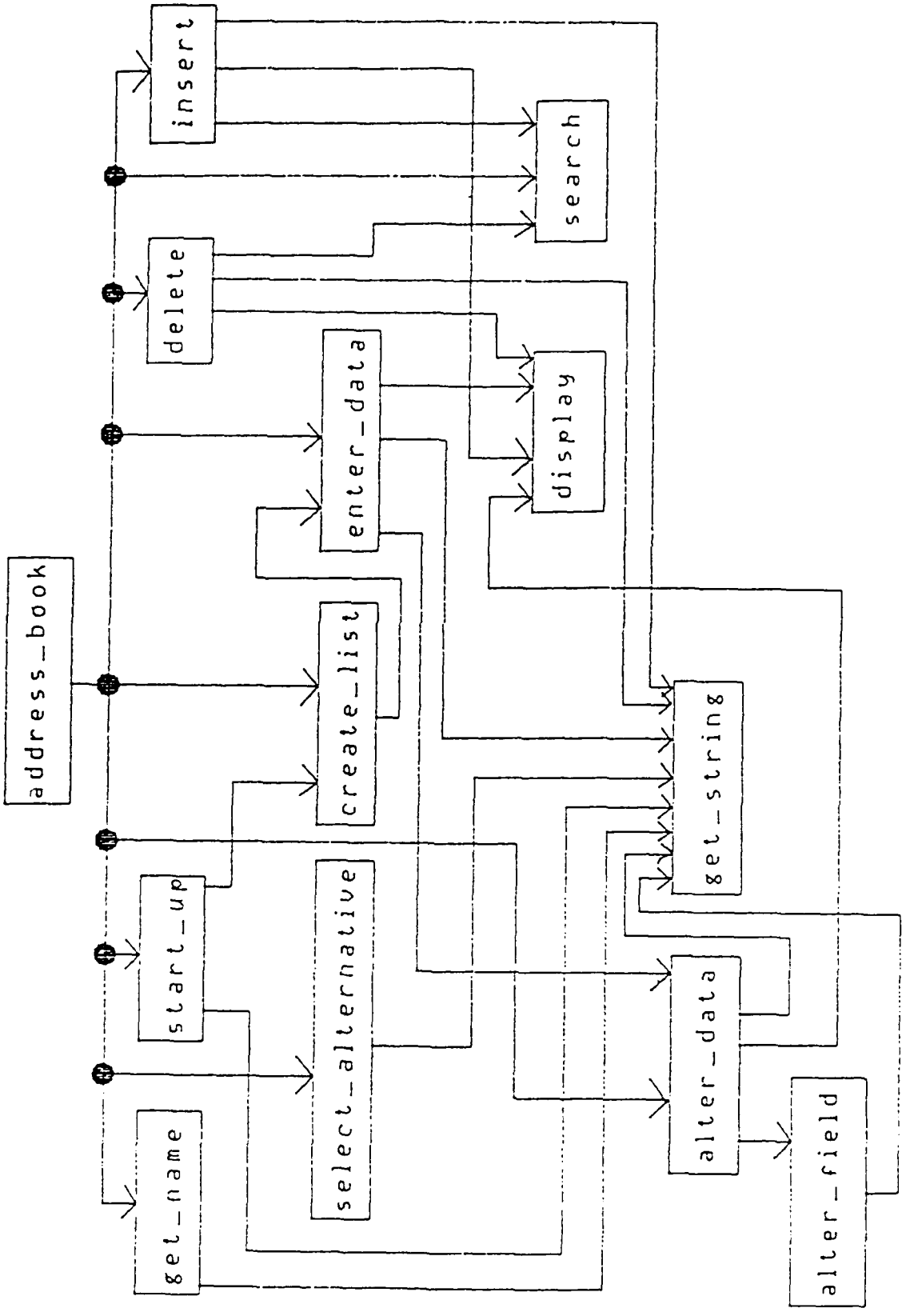
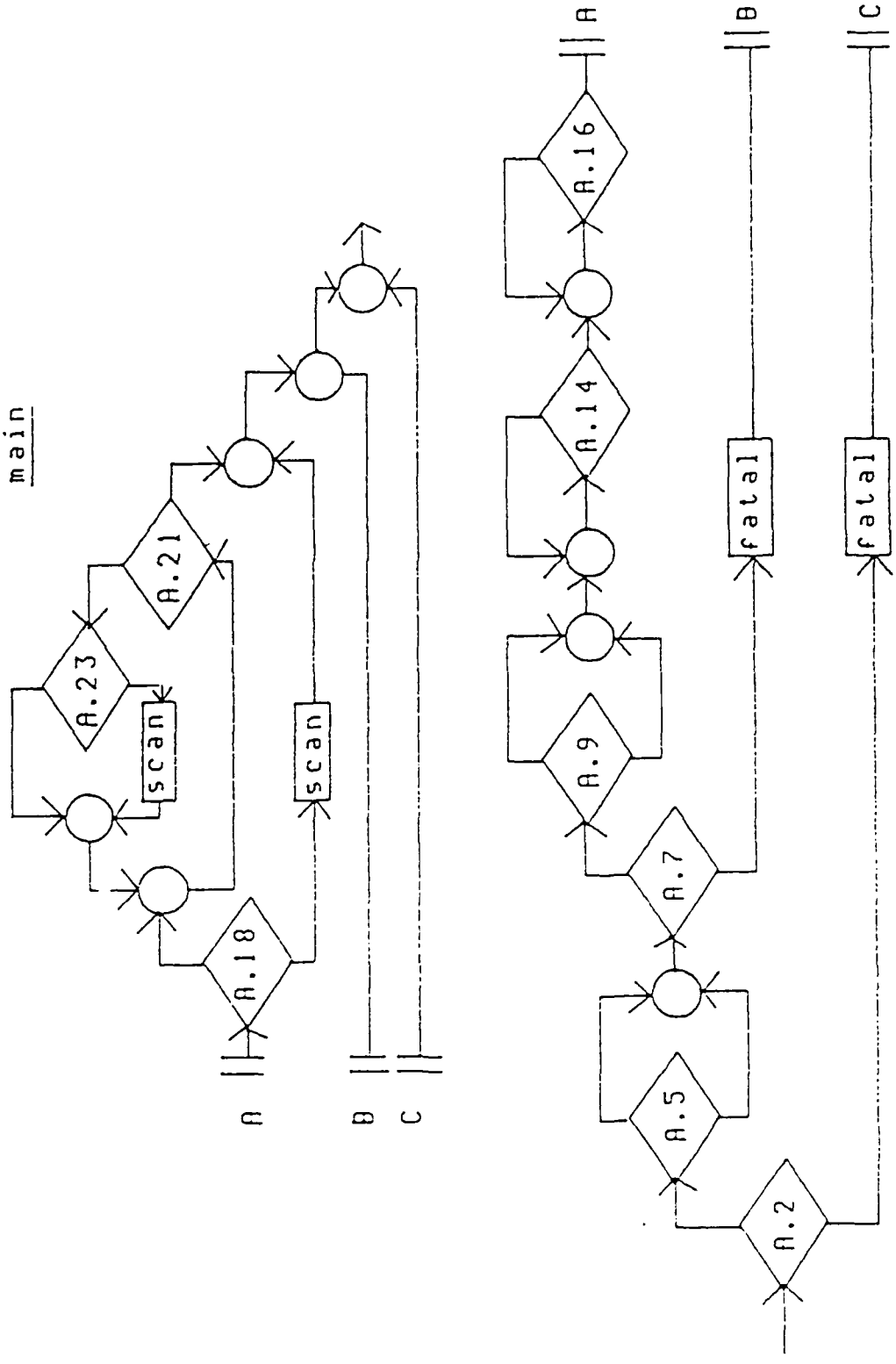


Figure 7. Execution Flow Chart for the Ada Program address.

APPENDIX F

MAXIMAL INTERSECT NUMBER (MIN) CHARTS



MIN = 6

Figure 8. MIN for the C Program fastfind.

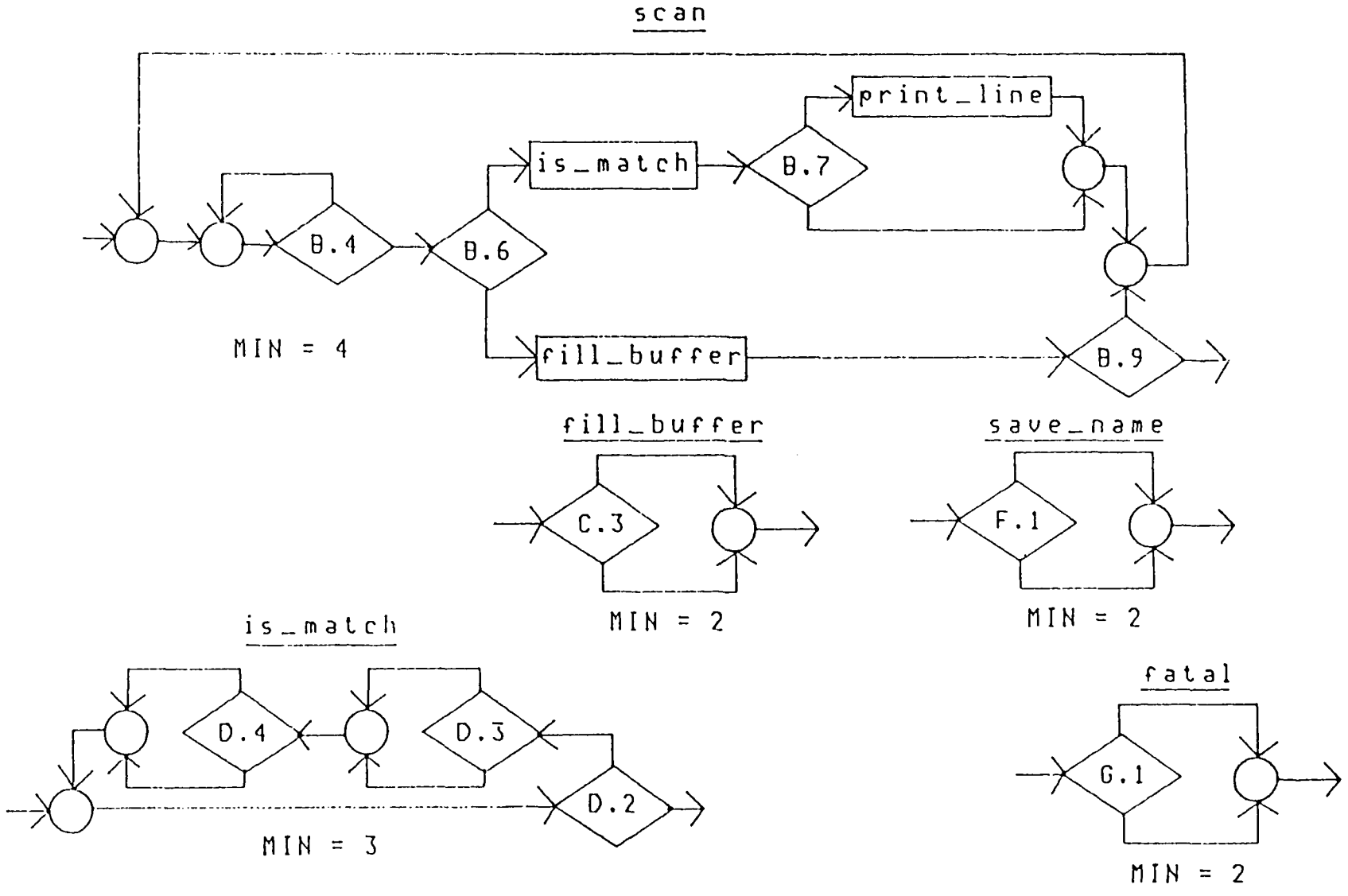
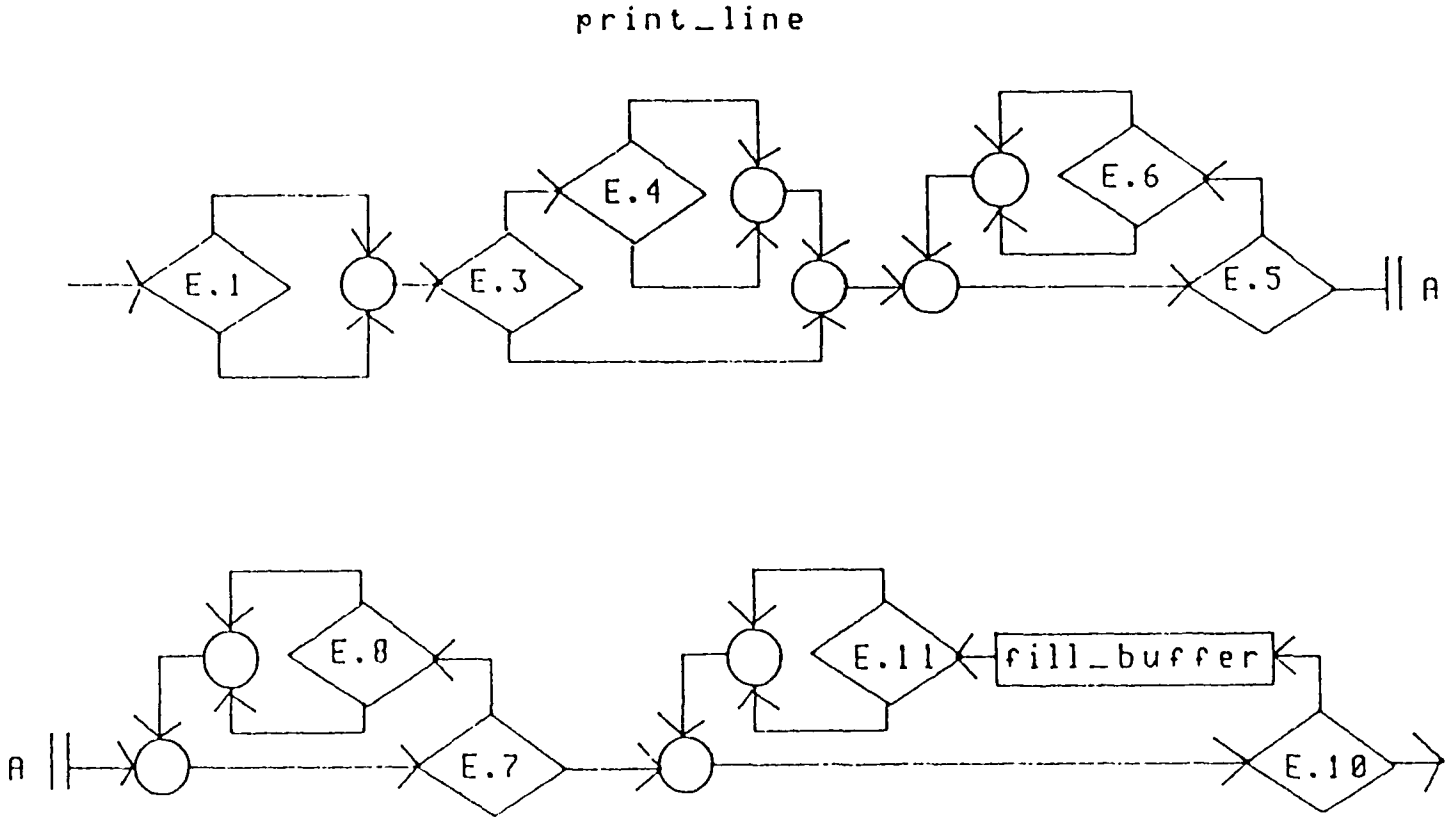


Figure 8. (Continued)



MIN = 2 + 3 + 3 + 3 + 3 - 2(5) + 2 = 6

Figure 8. (Continued)

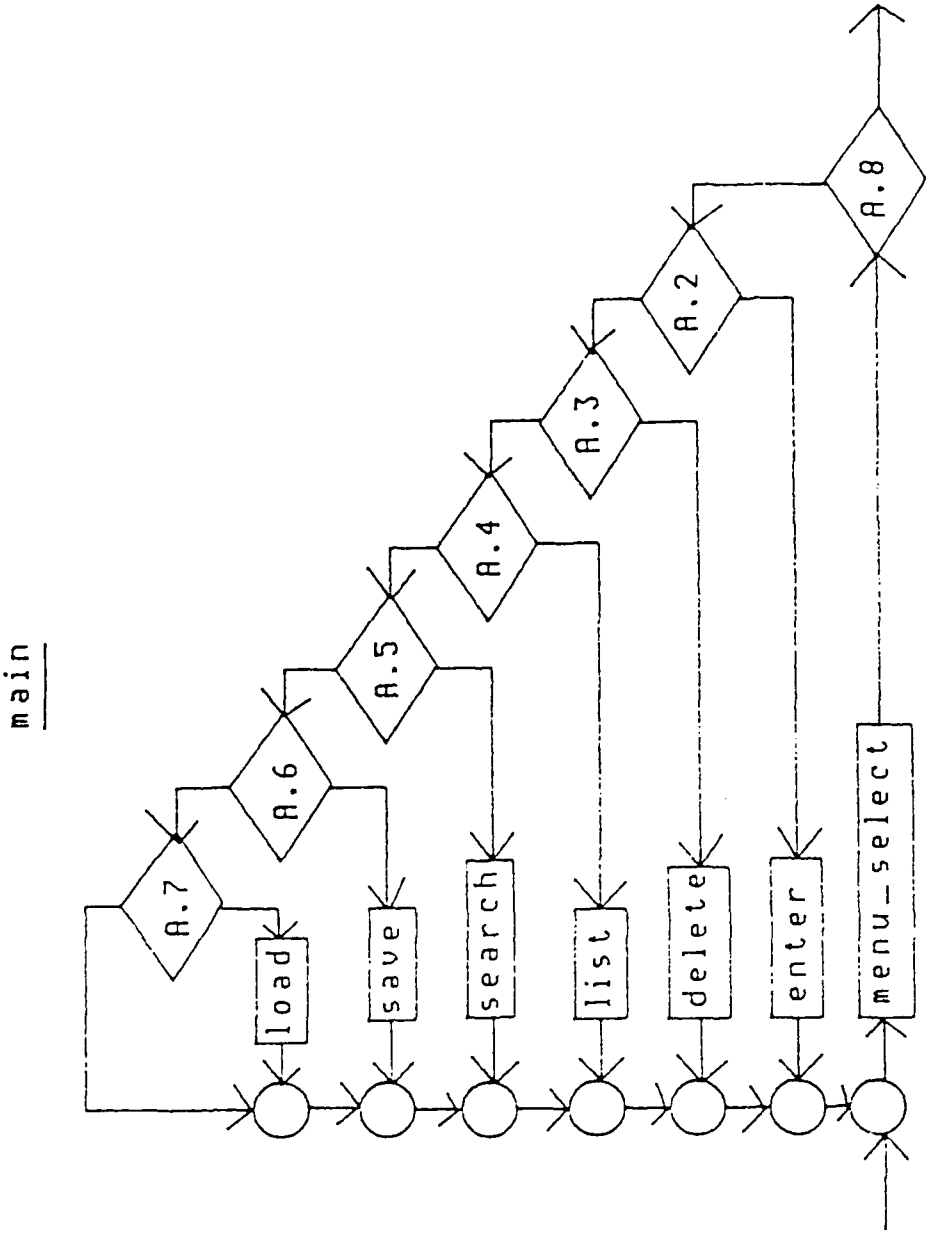


Figure 9. MIN for the C Program mail.

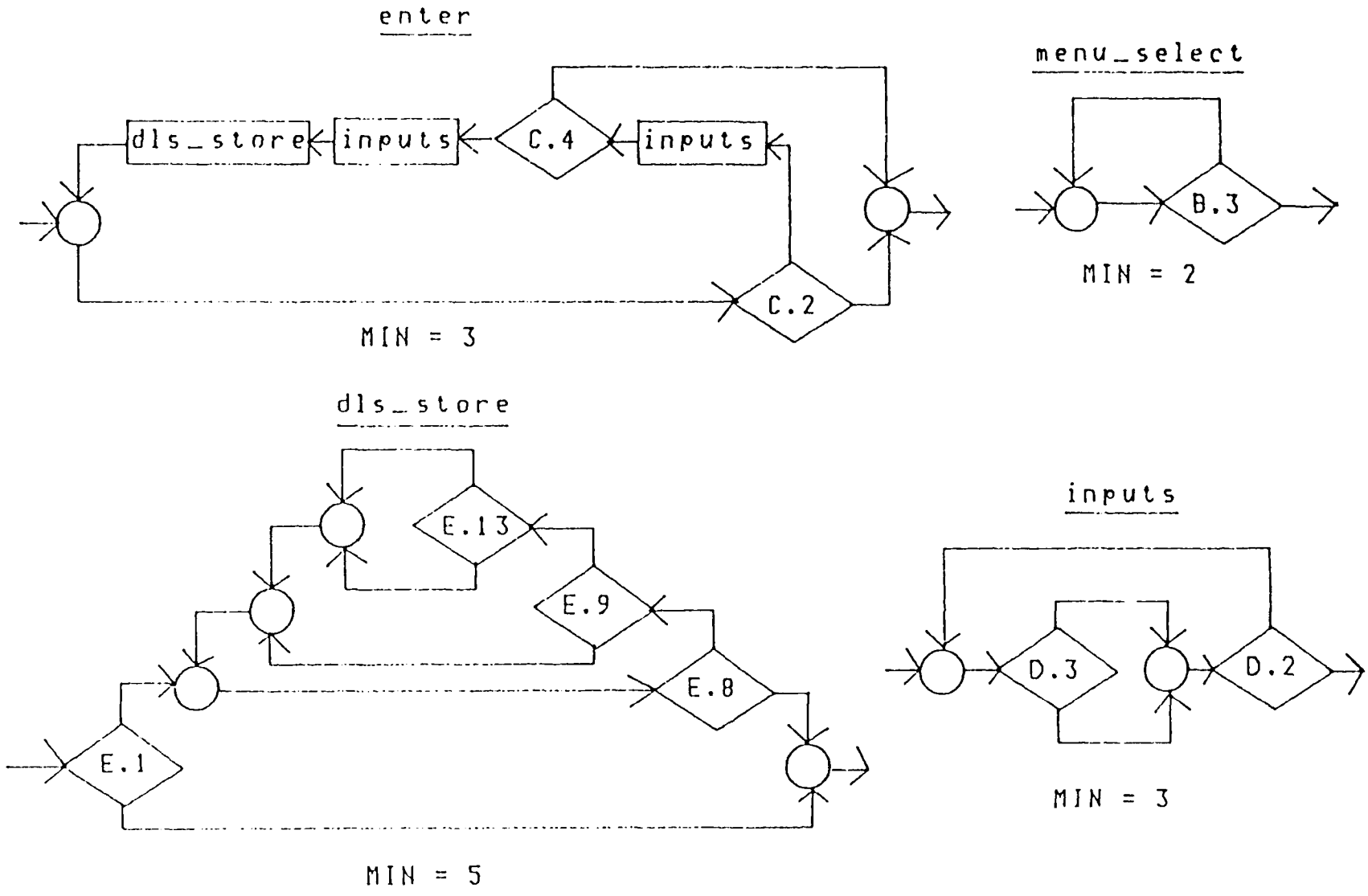


Figure 9. (Continued)

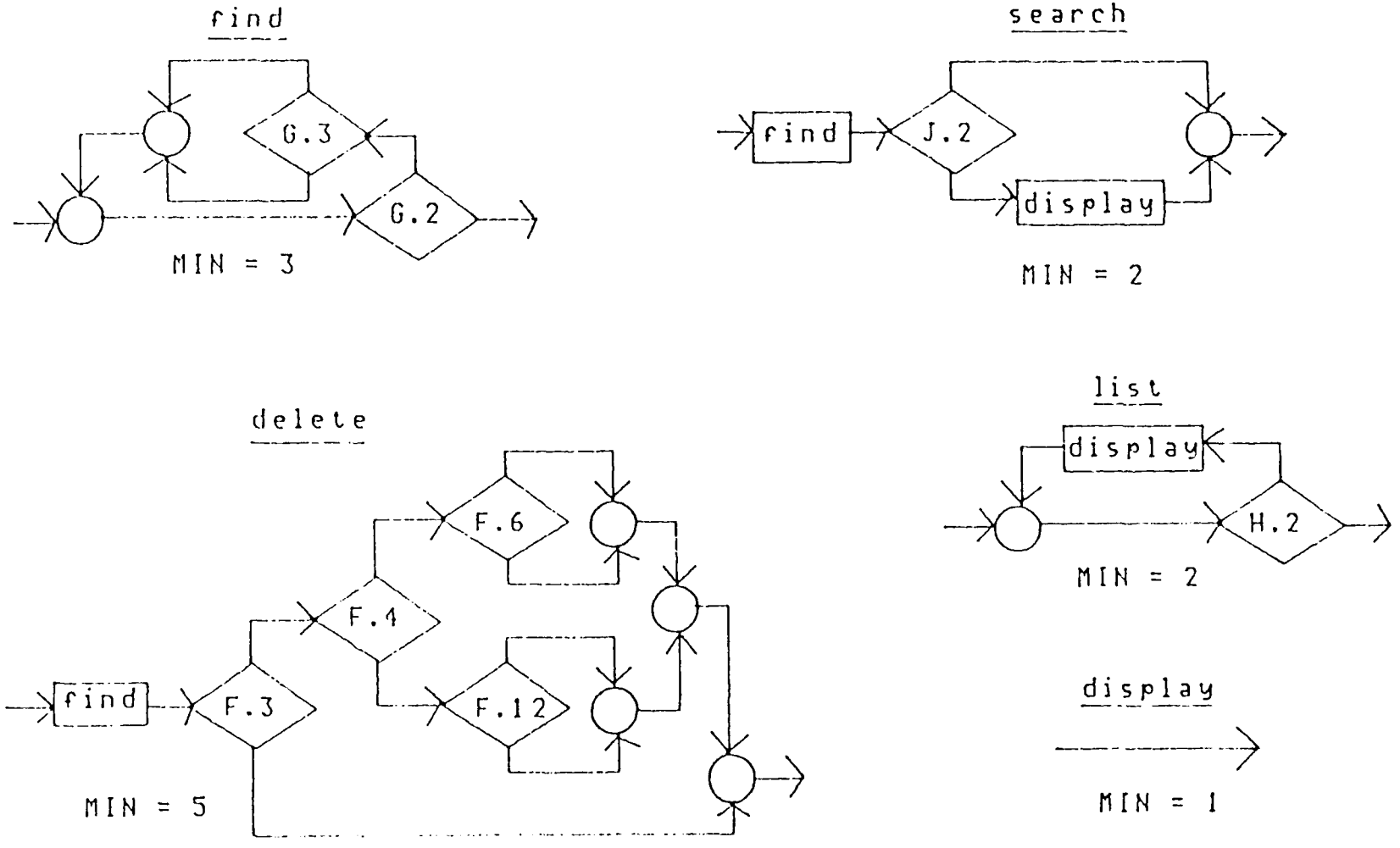


Figure 9. (Continued)

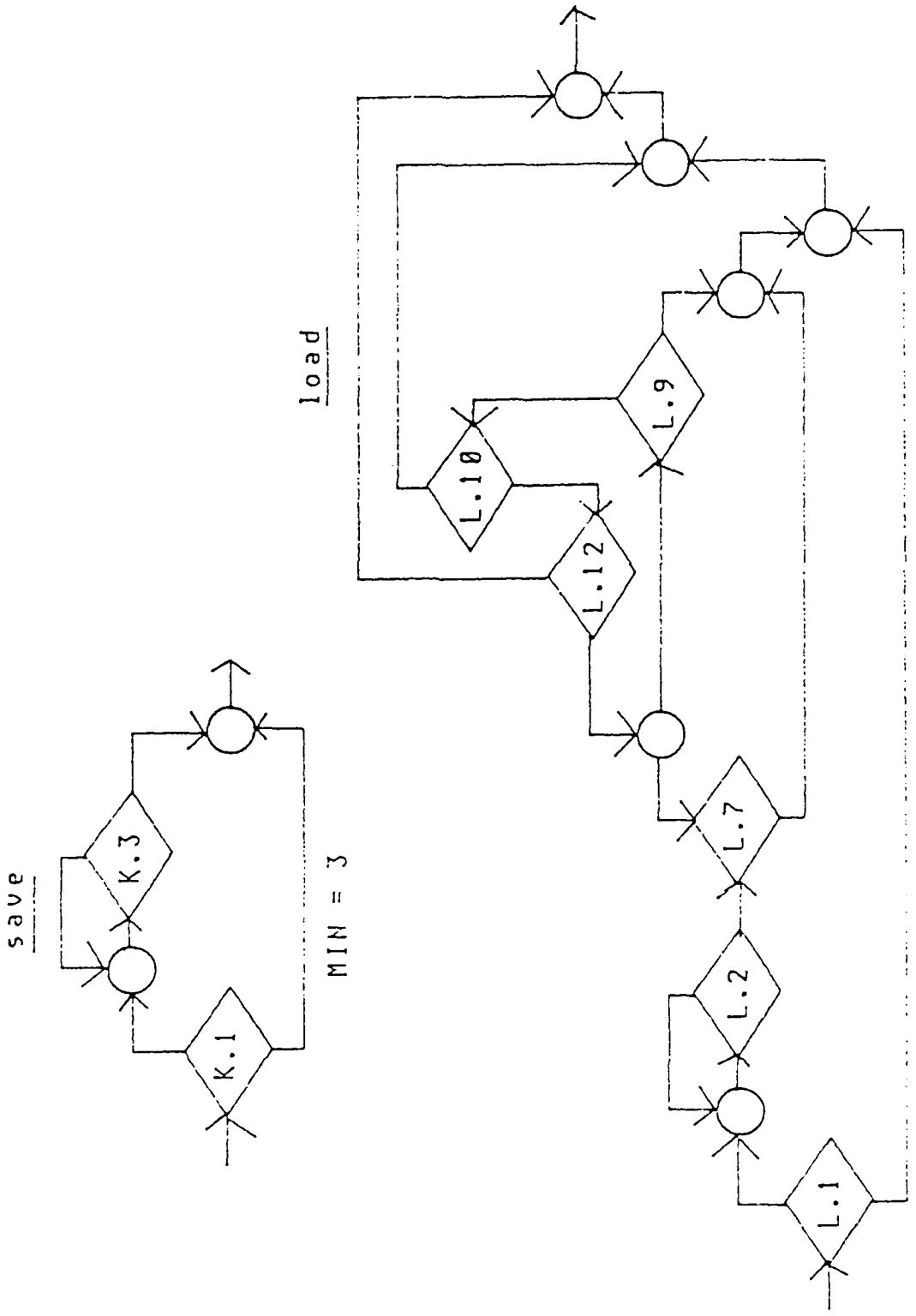


Figure 9. (Continued)

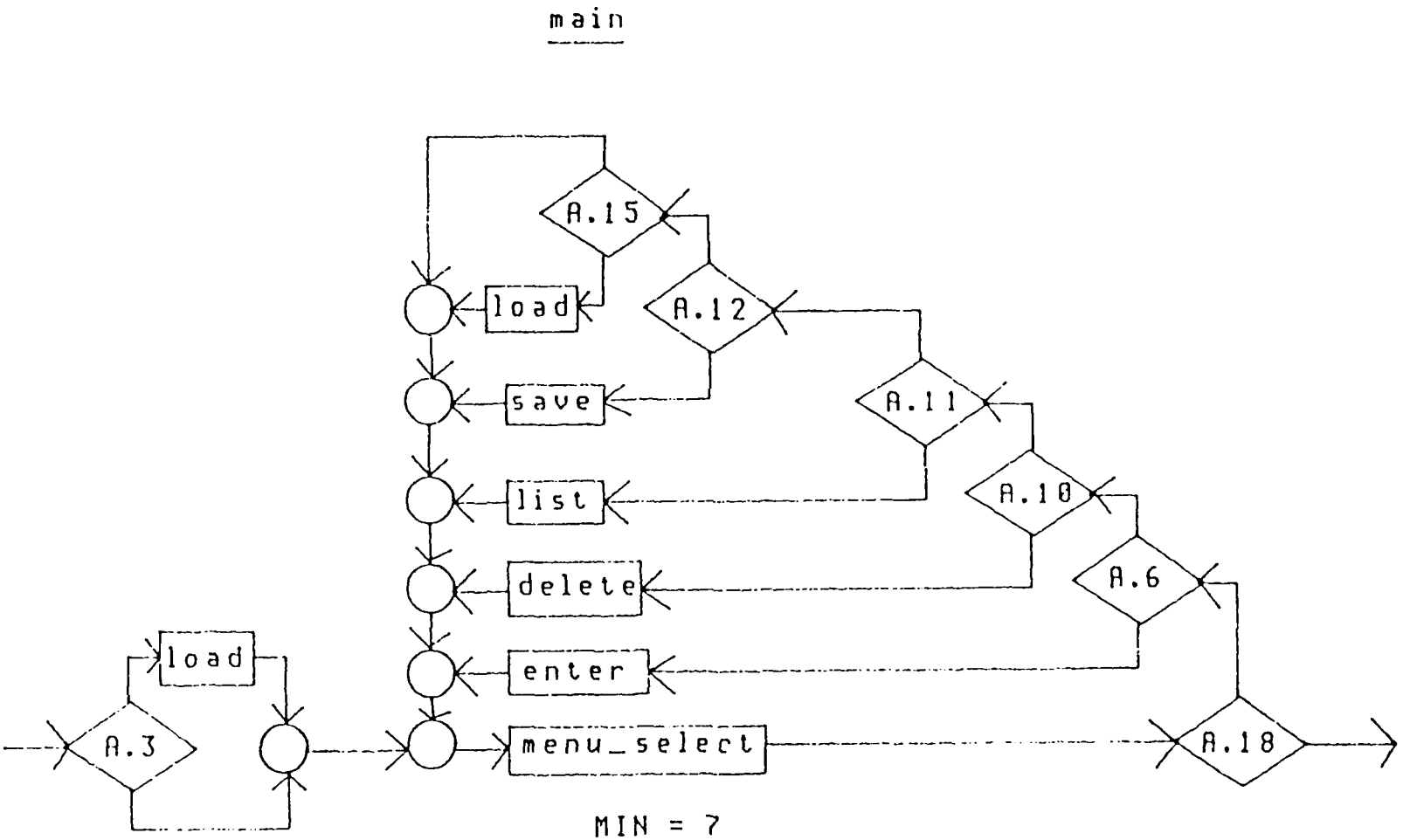


Figure 10. MIN for the C Program editor.

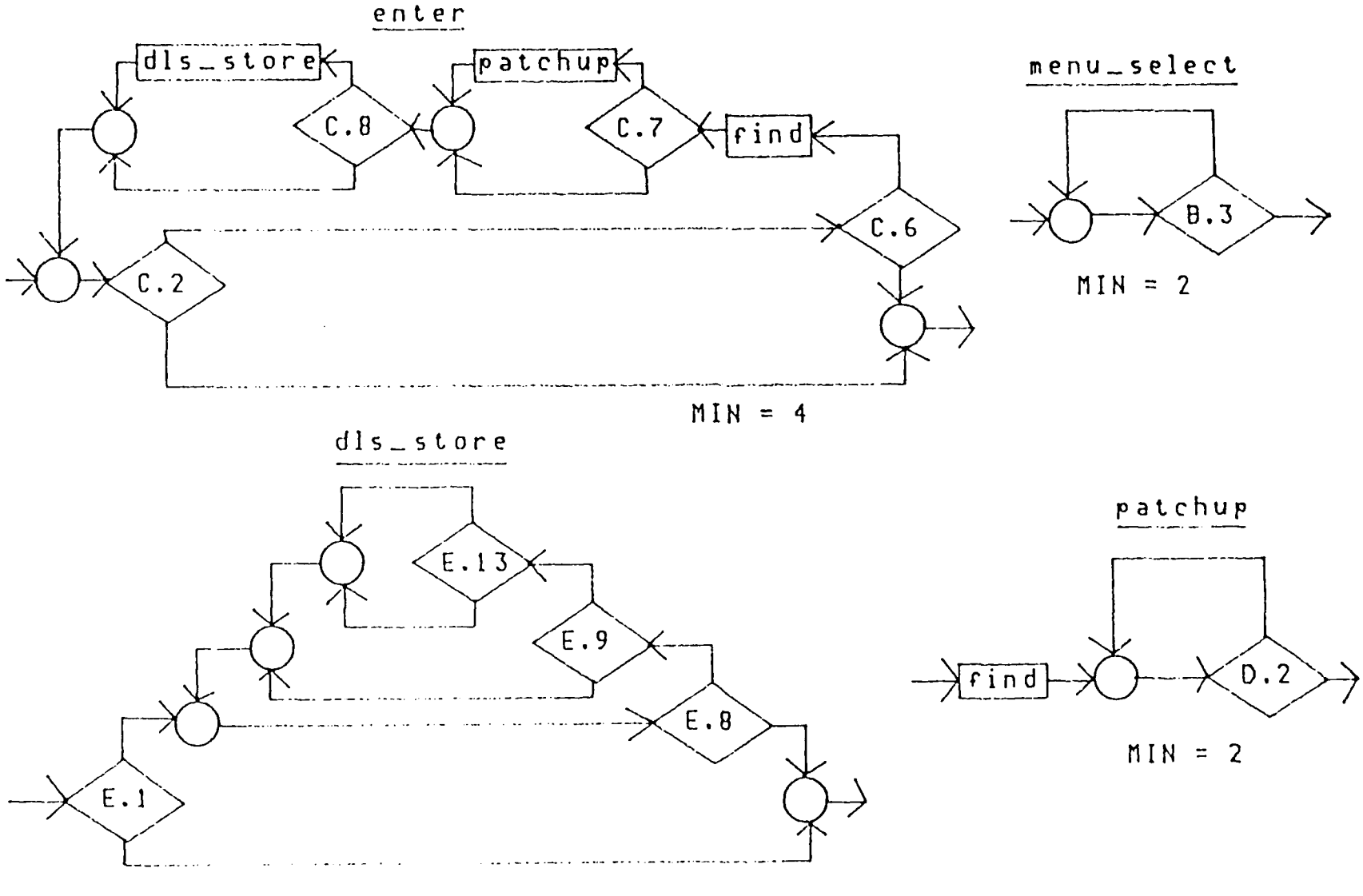
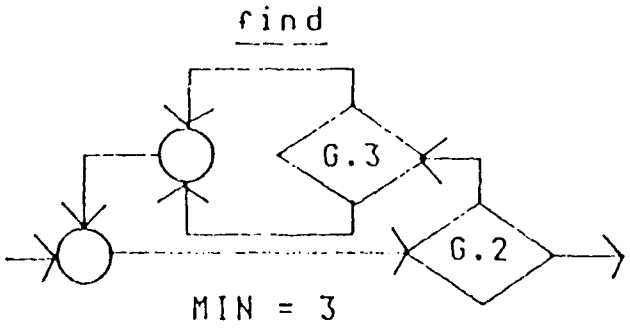
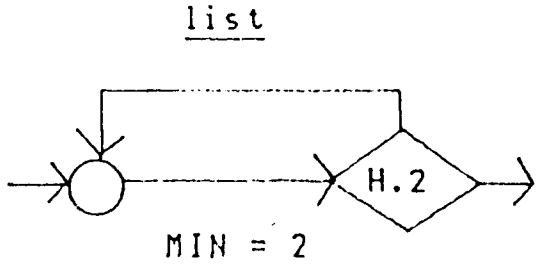


Figure 10. (Continued)





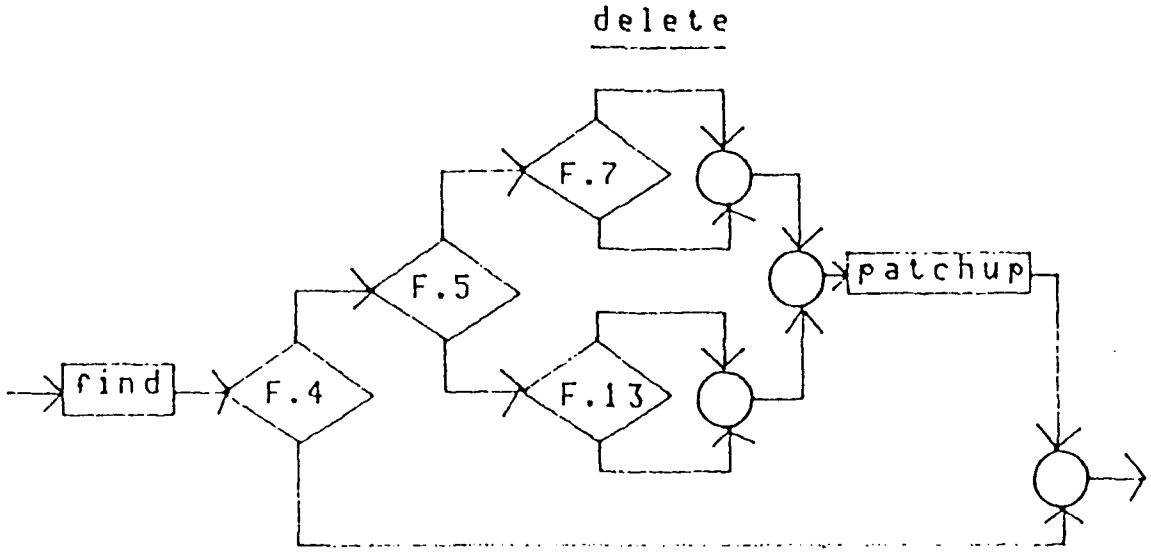


Figure 10. (Continued)

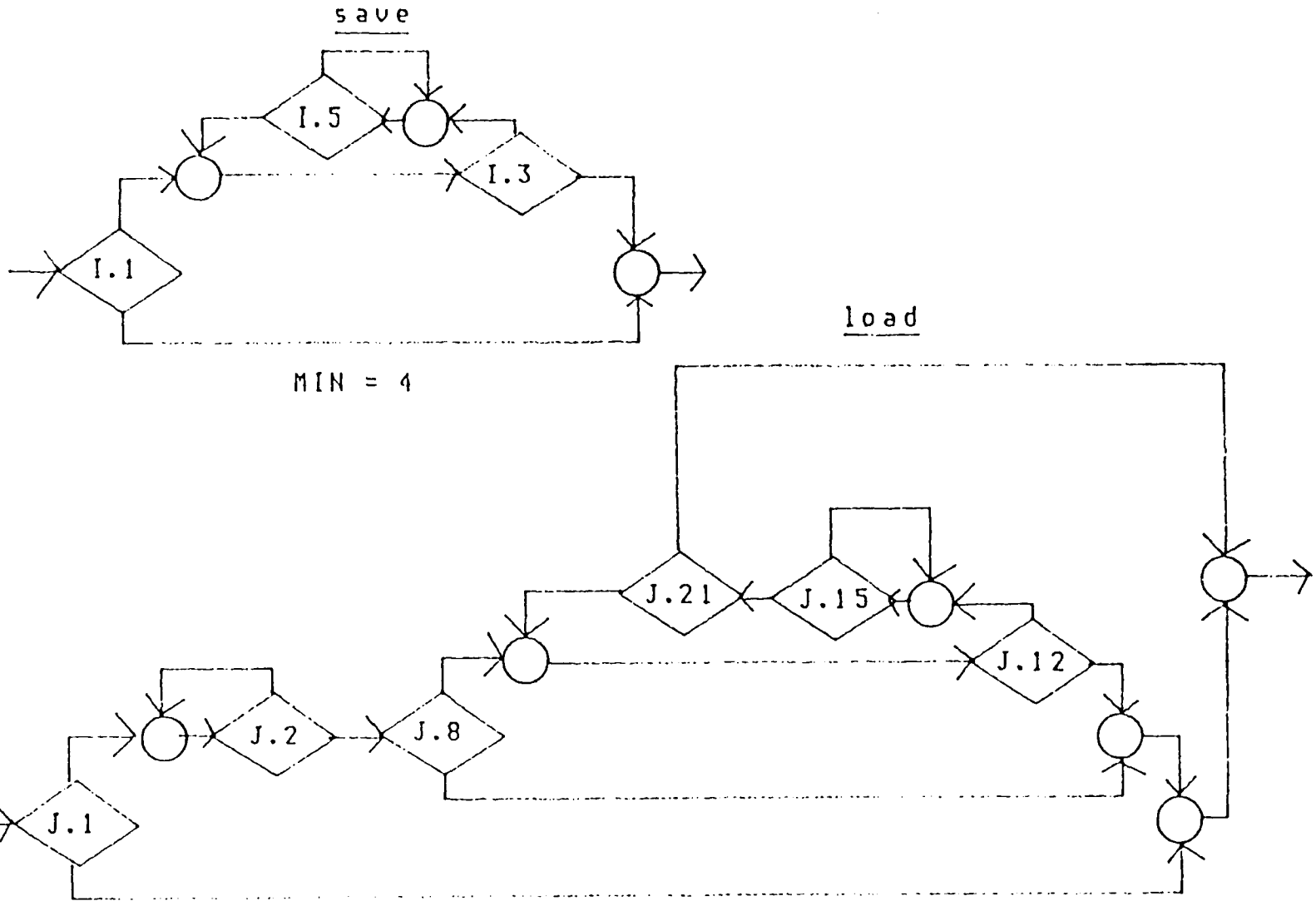
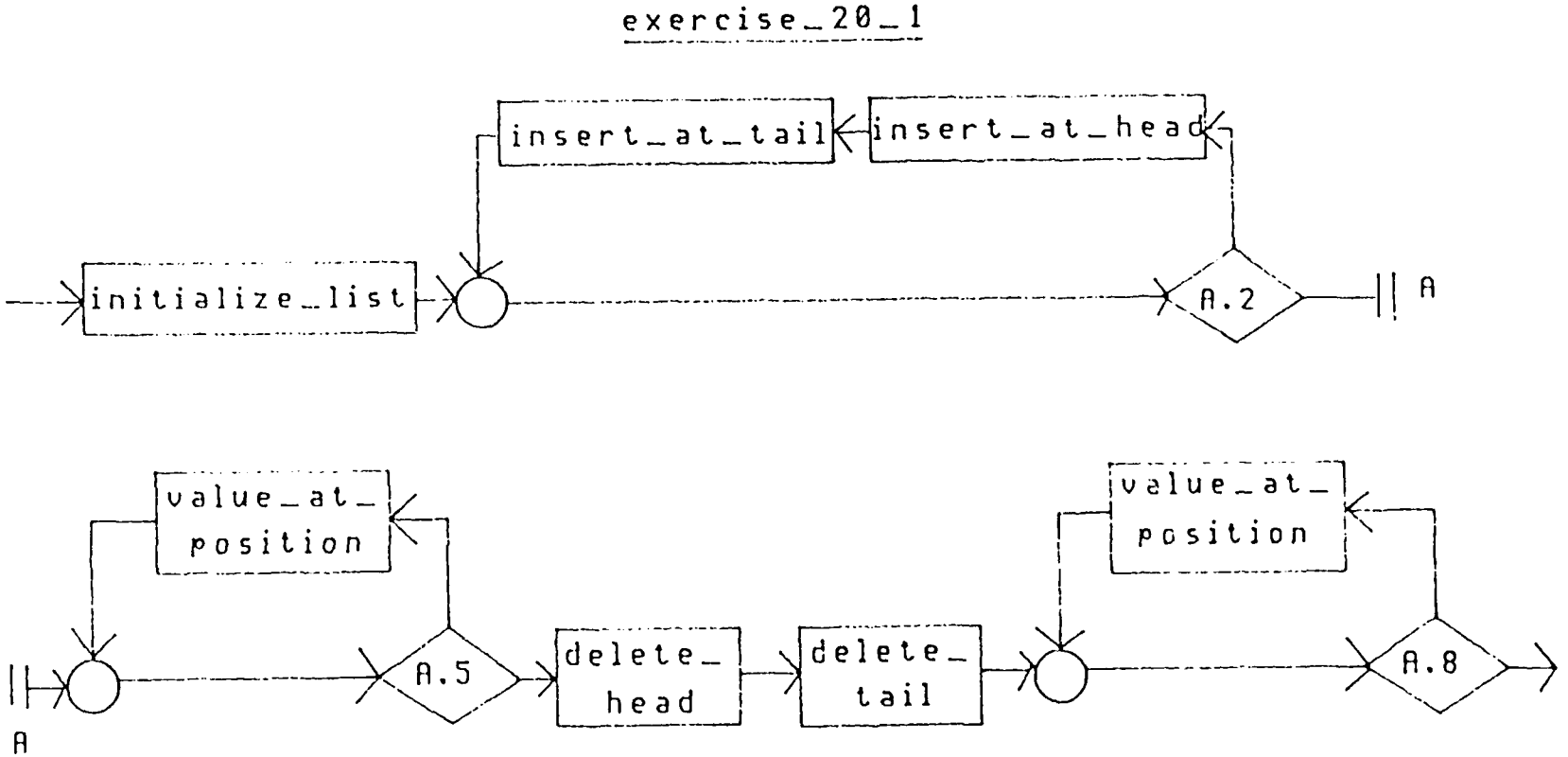


Figure 10. (Continued)



MIN = 2 + 2 + 2 - 2(3) + 2 = 2

Figure 11. MIN for the Ada Program intlist.

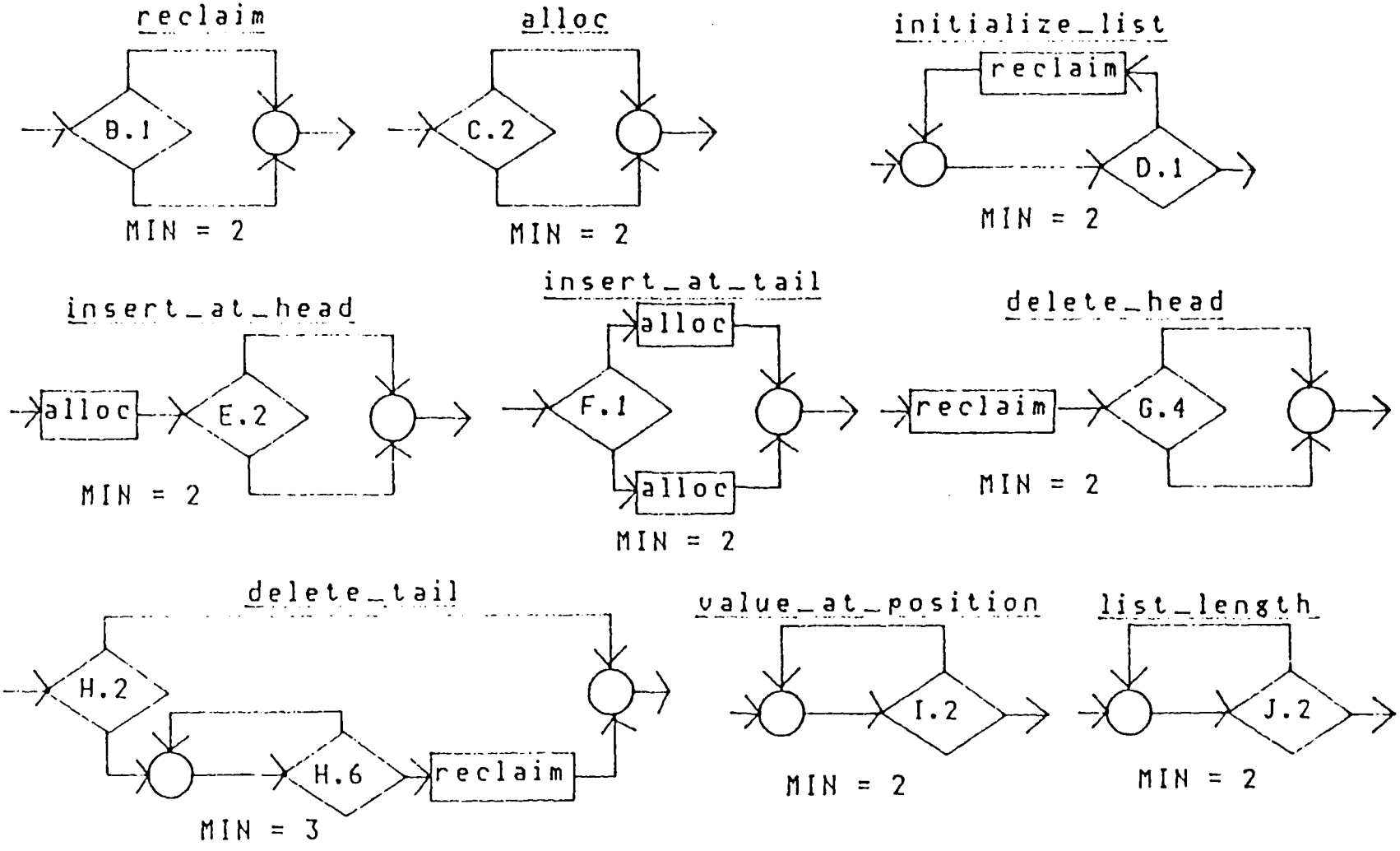


Figure 11. (Continued)

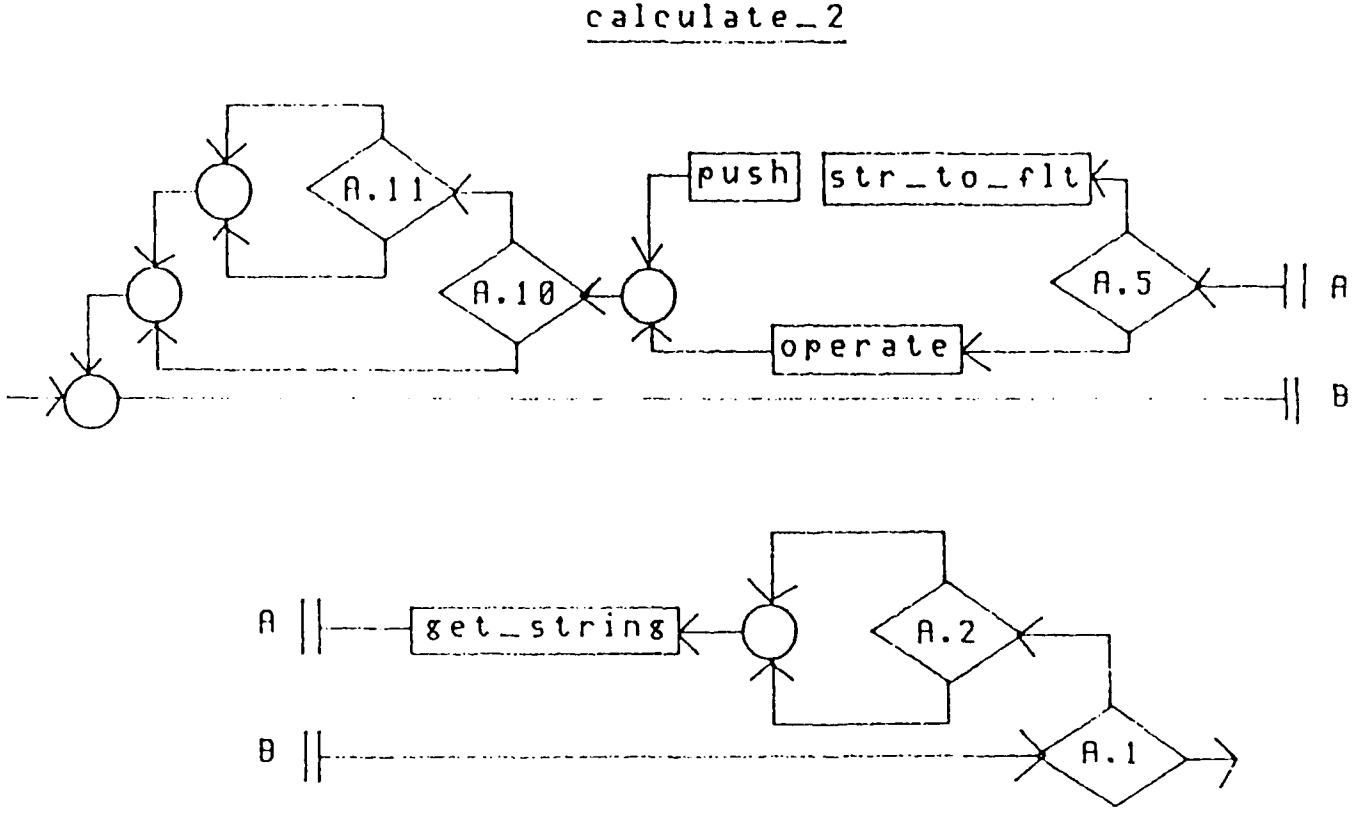


Figure 12. MIN for the Ada Program calc.

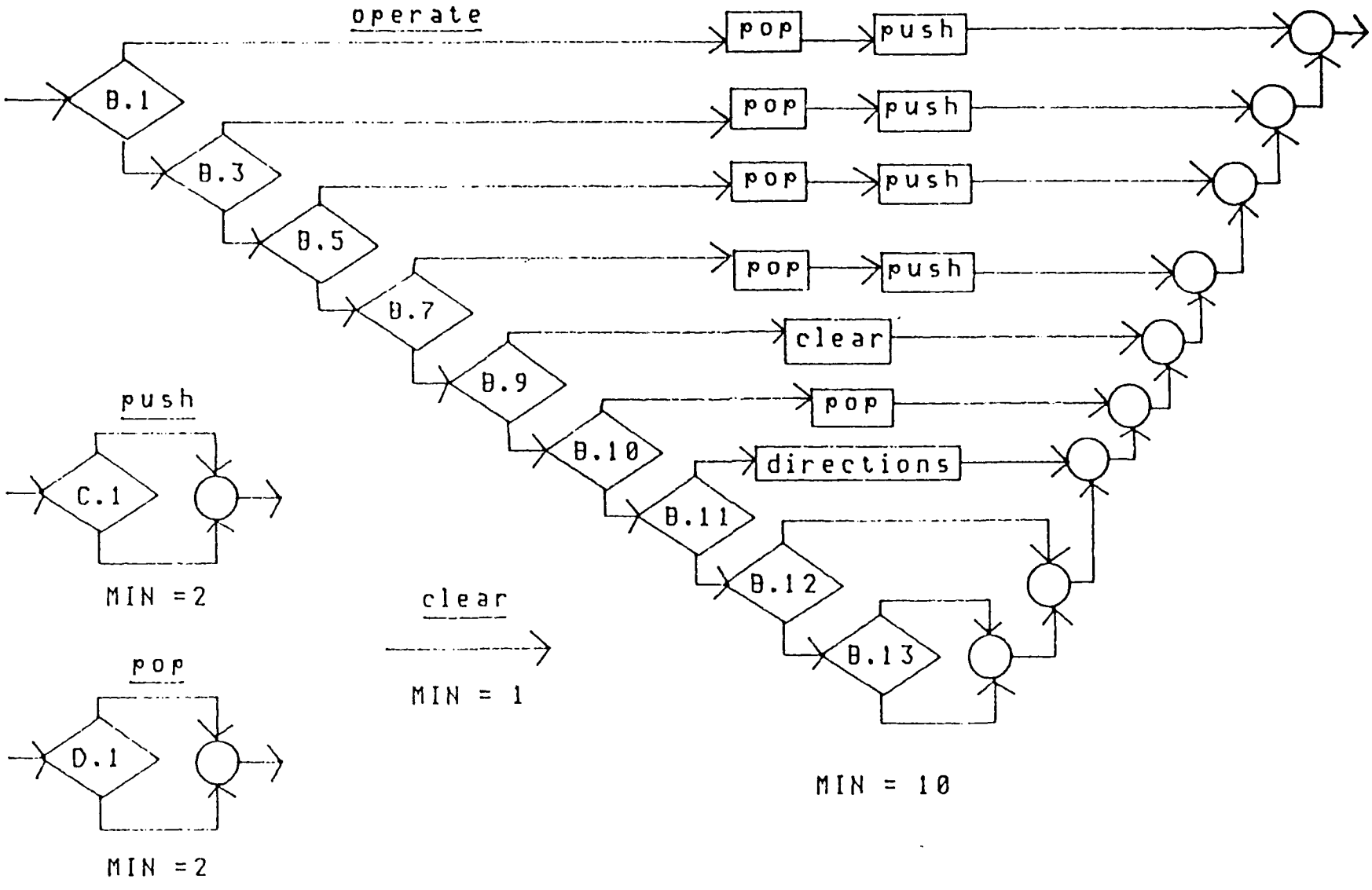


Figure 12. (Continued)

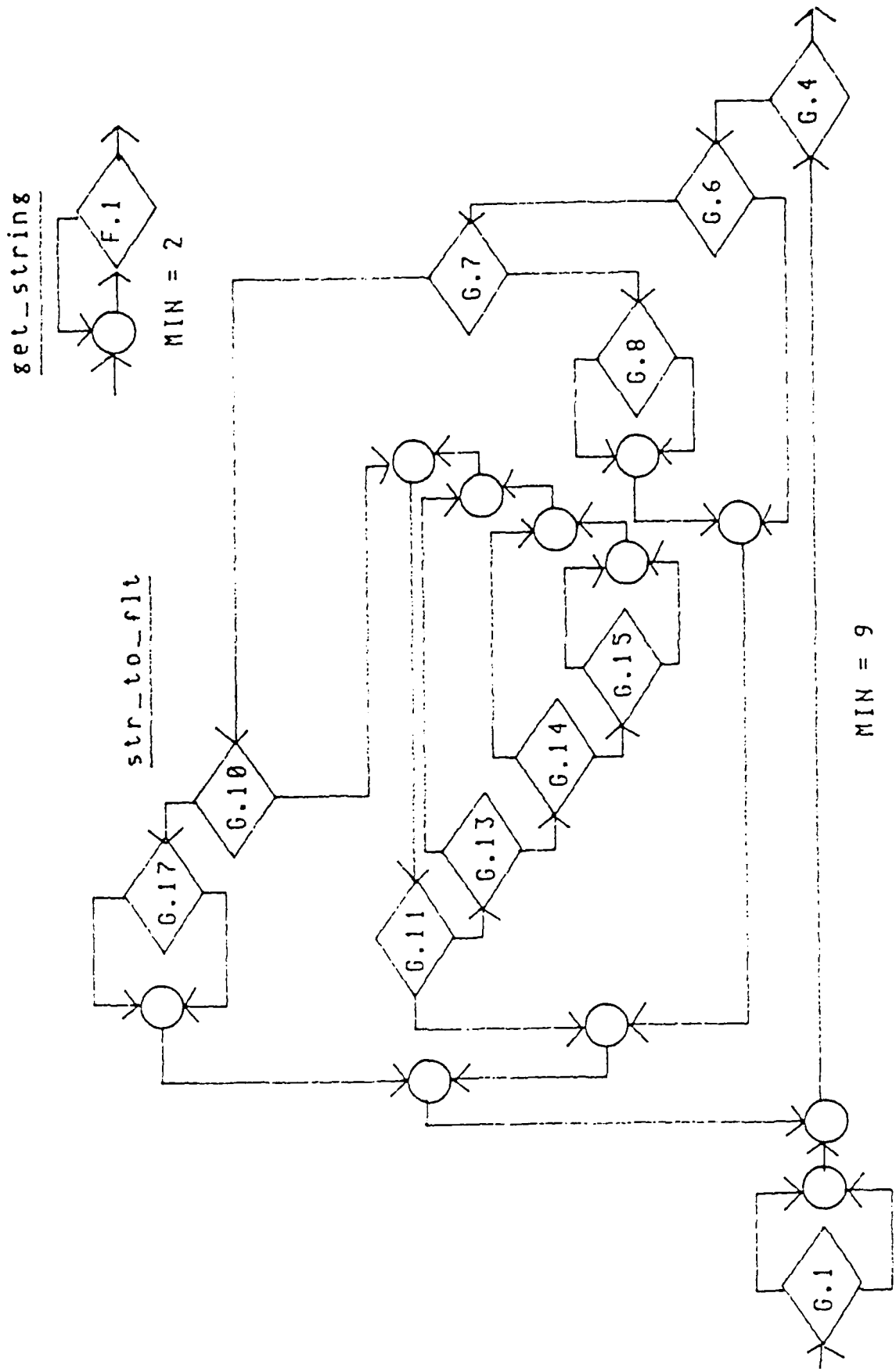


Figure 12. (Continued)

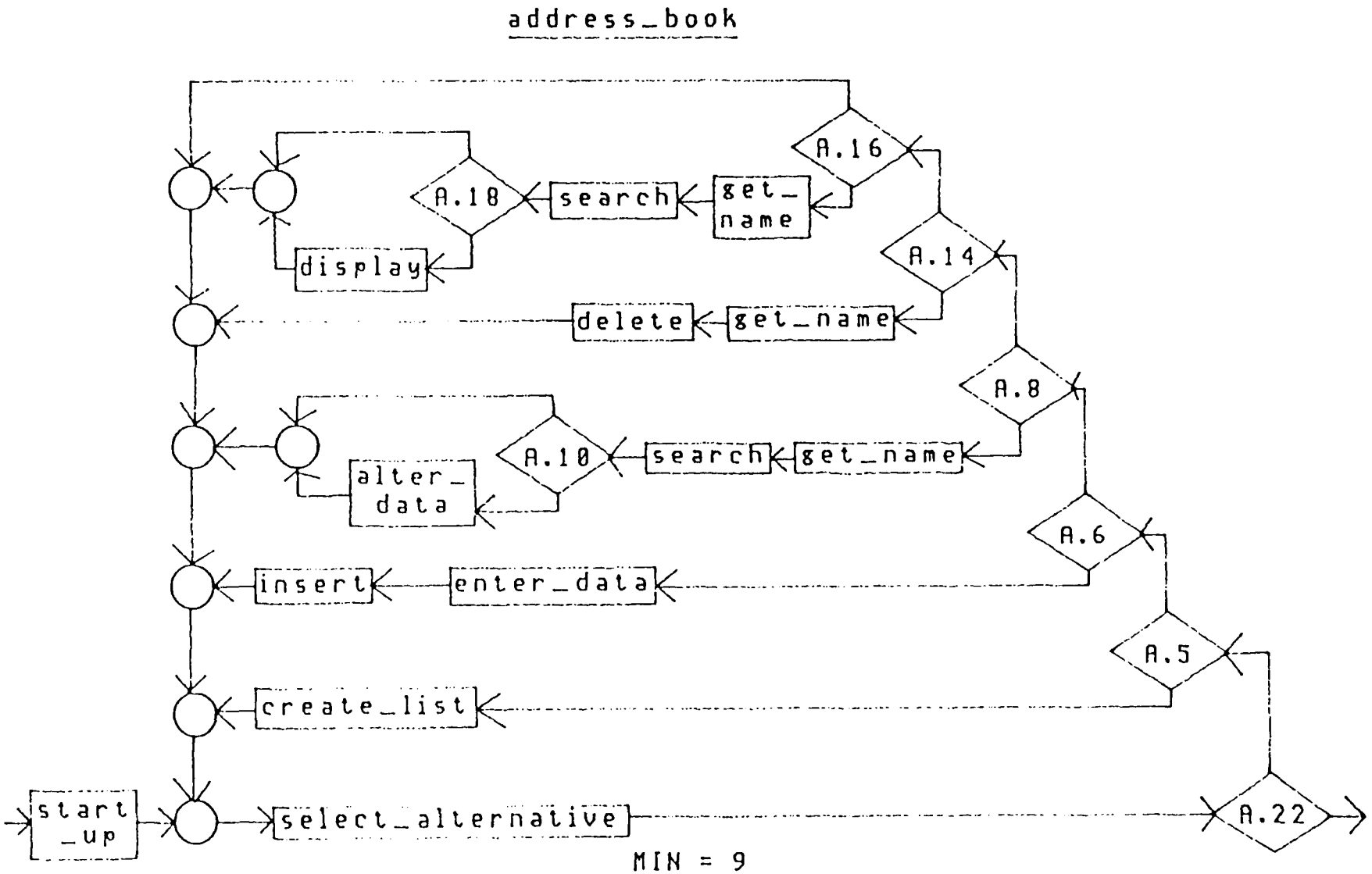


Figure 13. MIN for the Ada Program address.

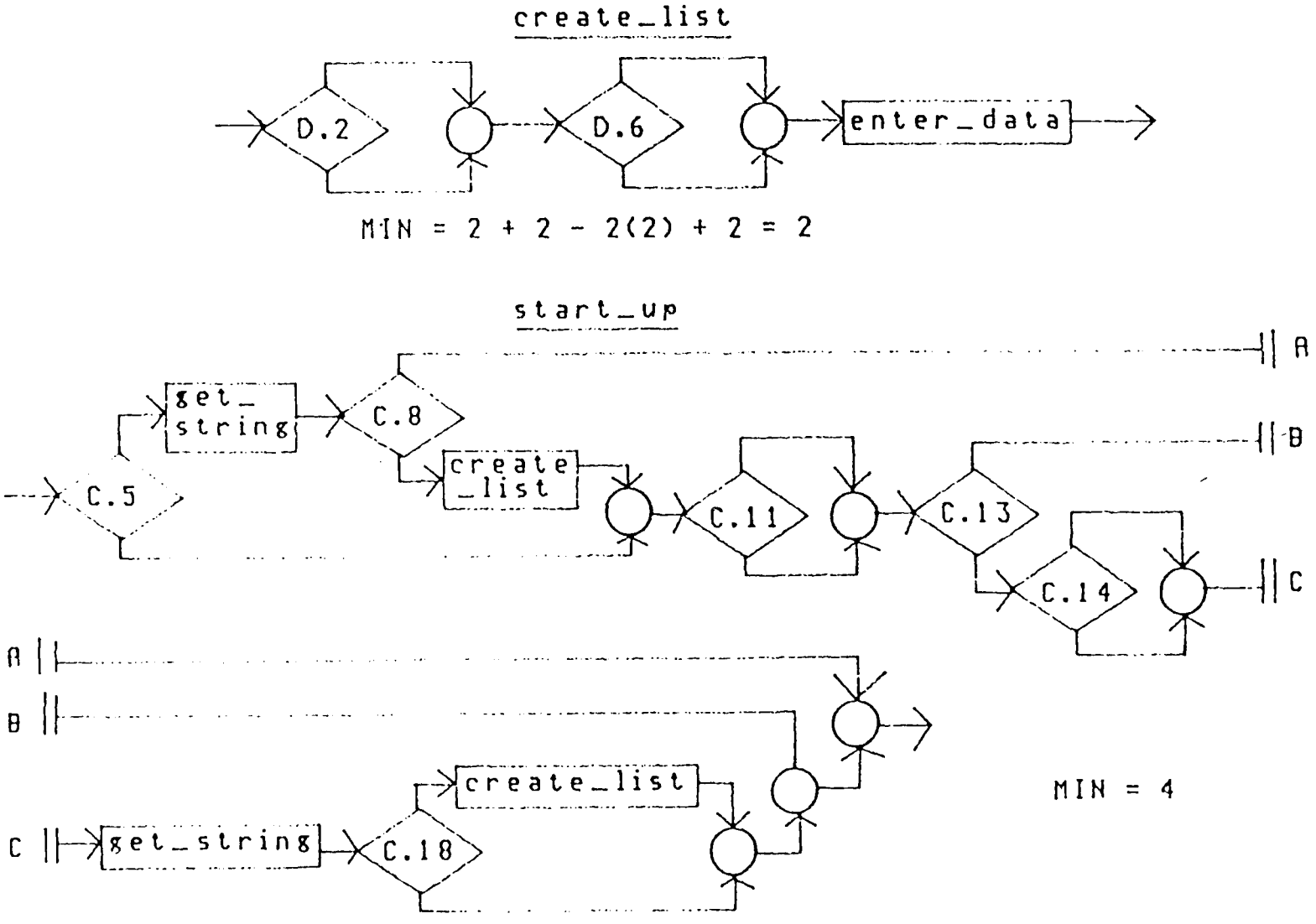


Figure 13. (Continued)

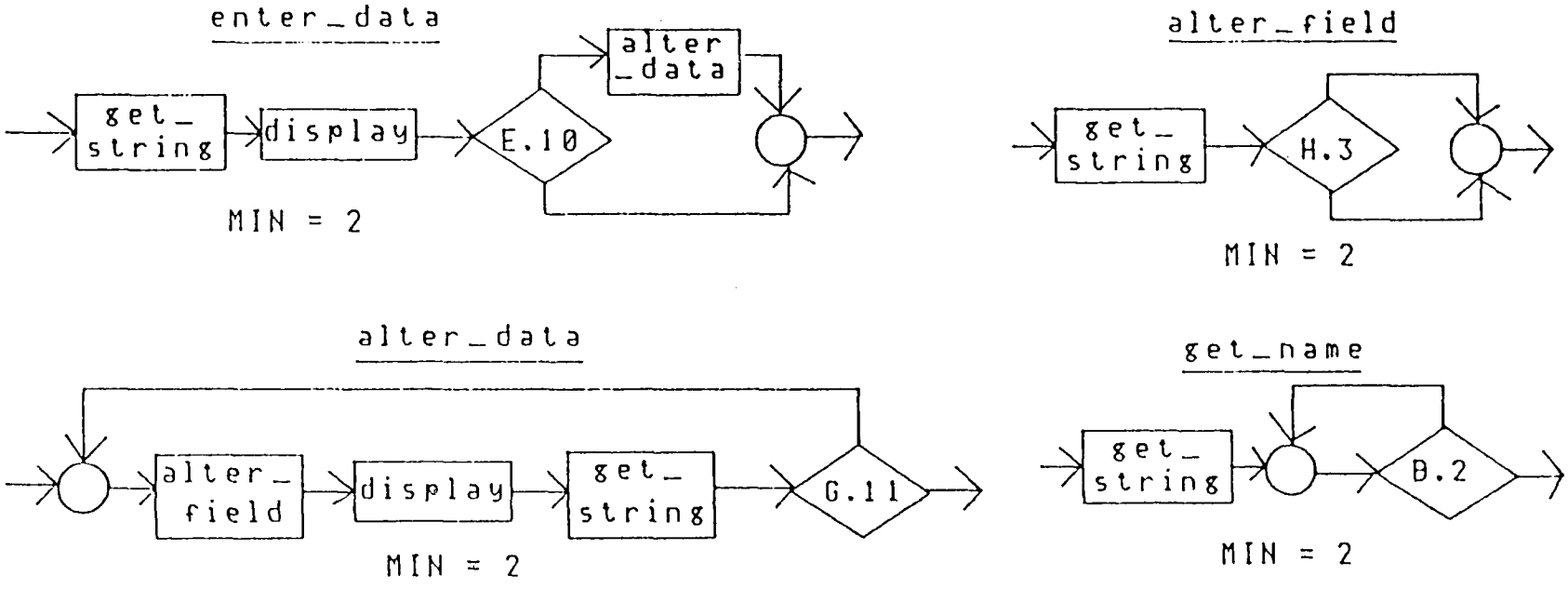
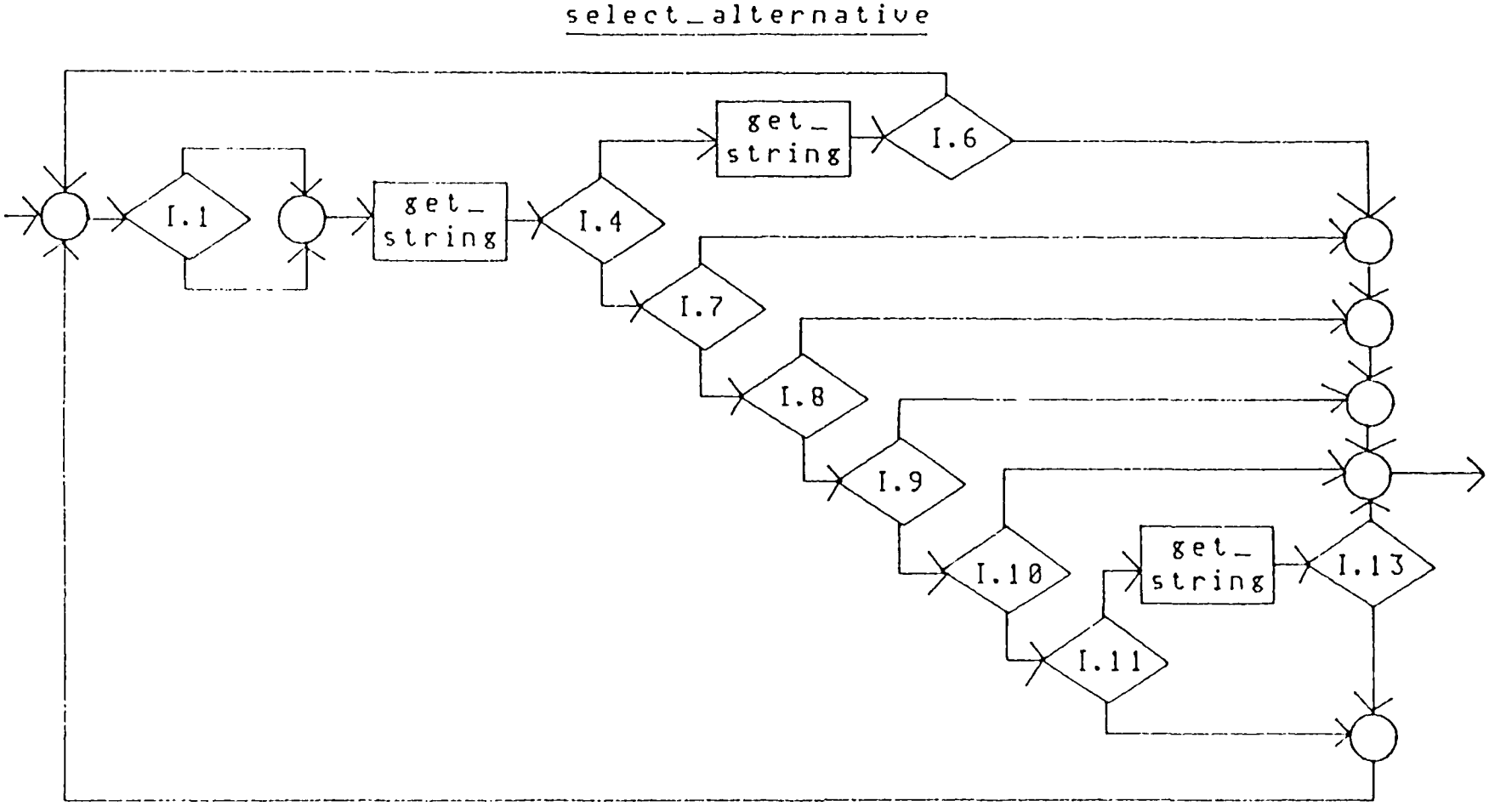


Figure 13. (Continued)



MIN = 9

Figure 13. (Continued)

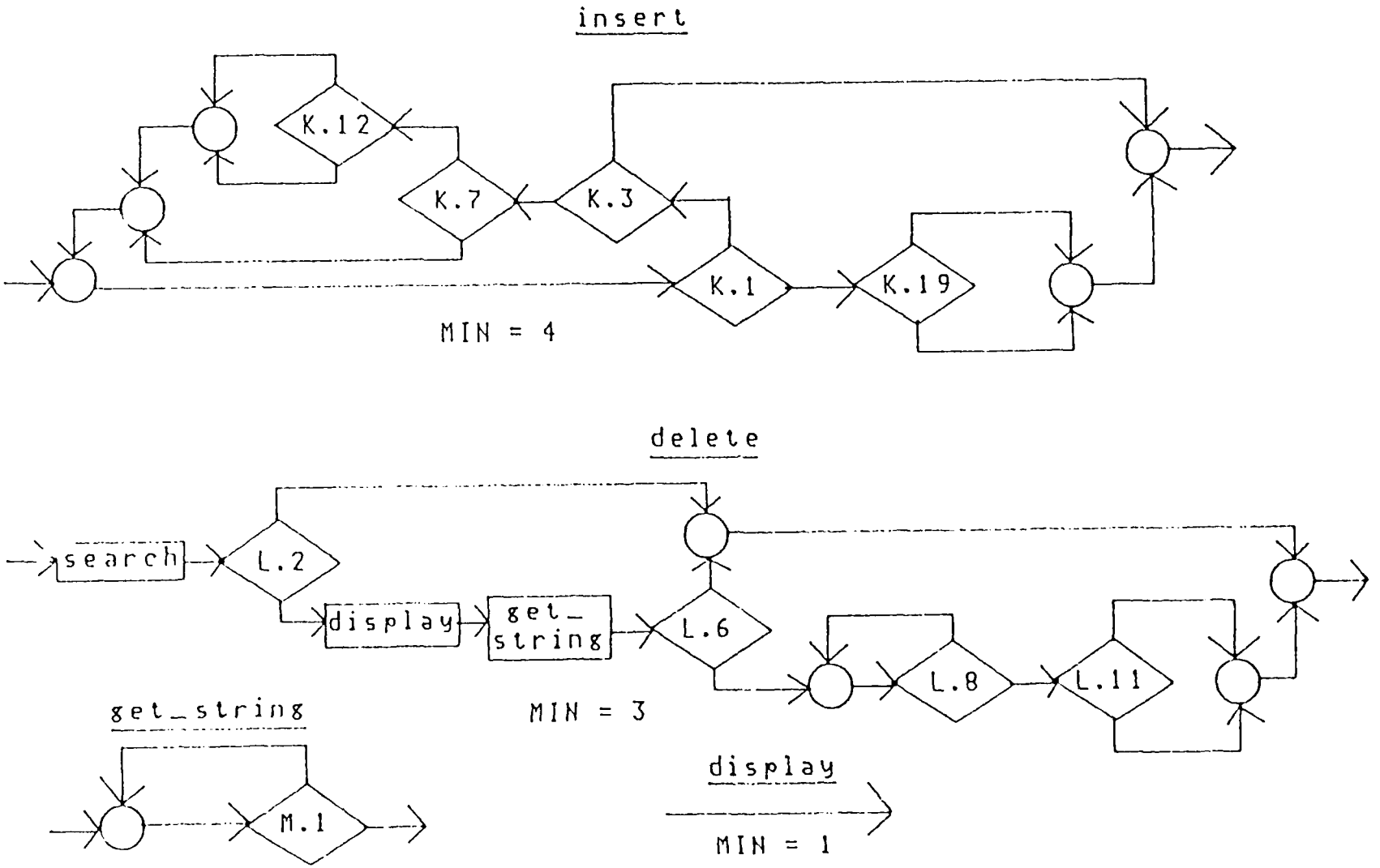


Figure 13. (Continued)

APPENDIX G

METRICS STATISTICAL DATA

CONTROL STRUCTURE ENTROPY
FOR ADA PROGRAMS

VARIABLE	N	MEAN	STD DEV	SUM	MINIMUM	MAXIMUM
LOCODE	9	4.643778	0.852238	41.79400	3.367300	6.200500
DECL	9	2.405444	1.256371	21.64900	0.000000	4.060400
NONDECL	9	4.511300	0.825977	40.60170	3.332200	6.075300
COMMENTS	9	3.162767	0.818810	28.46490	2.302600	4.844200
BLANK	9	2.260700	0.952463	20.34630	1.098600	4.077500
TOTAL	9	4.930589	0.835897	44.37530	3.784200	6.520600
CONSTENT	9	2.005511	1.190812	18.04960	0.000000	3.583500

PEARSON CORRELATION COEFFICIENTS / PROB > |R| UNDER H0:RHO=0 / N = 9

	LOCODE	DECL	NONDECL	COMMENTS	BLANK	TOTAL
CONSTENT	0.85509	0.83848	0.84668	0.69253	0.52251	0.81362
	0.0033	0.0047	0.0040	0.0387	0.1490	0.0076

PEARSON CORRELATION COEFFICIENTS / PROB > |R| UNDER HO:RHO=0 / N = 9

	LOCODE	DECL	NONDECL	COMMENTS	BLANK	TOTAL
LOCODE	1.00000	0.92663	0.99615	0.95139	0.87705	0.99628
	0.0000	0.0003	0.0001	0.0001	0.0019	0.0001
DECL	0.92663	1.00000	0.89244	0.83980	0.71795	0.90647
	0.0003	0.0000	0.0012	0.0046	0.0294	0.0008
NONDECL	0.99615	0.89244	1.00000	0.95445	0.88763	0.99513
	0.0001	0.0012	0.0000	0.0001	0.0014	0.0001
COMMENTS	0.95139	0.83980	0.95445	1.00000	0.94347	0.97284
	0.0001	0.0046	0.0001	0.0000	0.0001	0.0001
BLANK	0.87705	0.71795	0.88763	0.94347	1.00000	0.91169
	0.0019	0.0294	0.0014	0.0001	0.0000	0.0006
TOTAL	0.99628	0.90647	0.99513	0.97284	0.91169	1.00000
	0.0001	0.0008	0.0001	0.0001	0.0006	0.0000

OBS	LOCODE	DECL	NONDECL	COMMENTS	BLANK	TOTAL	CONSTENT
1	3.3673	0.0000	3.3322	2.3026	1.6094	3.7842	0.0000
2	4.1897	1.7918	4.0943	3.0445	2.3026	4.5747	0.6931
3	4.9200	2.3979	4.8363	3.1355	2.8332	5.1761	2.0794
4	3.8501	1.6094	3.7377	2.3979	1.0986	4.1109	1.0986
5	4.2047	1.7918	4.1109	2.3979	1.0986	4.3944	2.7726
6	5.1240	3.0910	4.9836	3.4965	2.3026	5.3519	3.1781
7	4.6444	3.2958	4.3438	2.9957	2.0794	4.8828	2.0794
8	5.2933	3.6109	5.0876	3.8501	2.9444	5.5797	2.5649
9	6.2005	4.0604	6.0753	4.8442	4.0775	6.5206	3.5835

ENTROPY LOADING METRIC
FOR ADA PROGRAMS

VARIABLE	N	MEAN	STD DEV	SUM	MINIMUM	MAXIMUM
LOCODE	9	4.643778	0.852238	41.79400	3.367300	6.200500
DECL	9	2.405444	1.256371	21.64900	0.000000	4.060400
NONDECL	9	4.511300	0.825977	40.60170	3.332200	6.075300
COMMENTS	9	3.162767	0.818810	28.46490	2.302600	4.844200
BLANK	9	2.260700	0.952463	20.34630	1.098600	4.077500
TOTAL	9	4.930589	0.835897	44.37530	3.784200	6.520600
OBJTOTAL	9	3.735656	0.767417	33.62090	2.833200	5.198500
AVOBEN	9	0.020089	0.368477	0.18080	-0.636200	0.536100
RATIO	9	-0.117589	0.378703	-1.05830	-0.710800	0.398500
SYSENT	9	1.590711	0.185535	14.31640	1.317600	1.929400
ENTLOA	9	3.618067	0.636825	32.56260	2.378000	4.487700

PEARSON CORRELATION COEFFICIENTS / PROB > |R| UNDER H0:RHO=0 / N = 9

	LOCODE	DECL	NONDECL	COMMENTS	BLANK	TOTAL
OBJTOTAL	0.98598	0.86925	0.98864	0.96155	0.92790	0.99210
	0.0001	0.0023	0.0001	0.0001	0.0003	0.0001
AVOBEN	-0.76924	-0.64946	-0.79343	-0.74274	-0.60339	-0.76190
	0.0154	0.0584	0.0107	0.0219	0.0854	0.0170
RATIO	-0.63680	-0.53980	-0.66186	-0.61173	-0.45289	-0.62573
	0.0651	0.1336	0.0522	0.0800	0.2209	0.0715
SYSENT	0.98521	0.93461	0.97448	0.95054	0.88982	0.98559
	0.0001	0.0002	0.0001	0.0001	0.0013	0.0001
ENTLOA	0.80946	0.72646	0.79777	0.79493	0.84885	0.82342
	0.0082	0.0266	0.0100	0.0105	0.0038	0.0064

PEARSON CORRELATION COEFFICIENTS / PROB > |R| UNDER H0:RHO=0 / N = 9

	LOCODE	DECL	NONDECL	COMMENTS	BLANK	TOTAL
LOCODE	1.00000	0.92663	0.99615	0.95139	0.87705	0.99628
	0.0000	0.0003	0.0001	0.0001	0.0019	0.0001
DECL	0.92663	1.00000	0.89244	0.83980	0.71795	0.90647
	0.0003	0.0000	0.0012	0.0046	0.0294	0.0008
NONDECL	0.99615	0.89244	1.00000	0.95445	0.88763	0.99513
	0.0001	0.0012	0.0000	0.0001	0.0014	0.0001
COMMENTS	0.95139	0.83980	0.95445	1.00000	0.94347	0.97284
	0.0001	0.0046	0.0001	0.0000	0.0001	0.0001
BLANK	0.87705	0.71795	0.88763	0.94347	1.00000	0.91169
	0.0019	0.0294	0.0014	0.0001	0.0000	0.0006
TOTAL	0.99628	0.90647	0.99513	0.97284	0.91169	1.00000
	0.0001	0.0008	0.0001	0.0001	0.0006	0.0000

PEARSON CORRELATION COEFFICIENTS / PROB > |R| UNDER H0:RHO=0 / N = 9

	OBJTOTAL	AVOBEN	RATIO	SYSENT	ENTLOA
OBJTOTAL	1.00000	-0.71176	-0.56219	0.97905	0.87073
	0.0000	0.0315	0.1151	0.0001	0.0022
AVOBEN	-0.71176	1.00000	0.97994	-0.67906	-0.27498
	0.0315	0.0000	0.0001	0.0443	0.4739
RATIO	-0.56219	0.97994	1.00000	-0.53760	-0.08282
	0.1151	0.0001	0.0000	0.1355	0.8322
SYSENT	0.97905	-0.67906	-0.53760	1.00000	0.86010
	0.0001	0.0443	0.1355	0.0000	0.0029
ENTLOA	0.87073	-0.27498	-0.08282	0.86010	1.00000
	0.0022	0.4739	0.8322	0.0029	0.0000

ENTROPY LOADING METRIC
FOR ADA PROGRAMS

OBS	LOCODE	DECL	NONDECL	COMMENTS	BLANK	TOTAL	OBJTOTAL	AVOBEN	RATIO	SYSENT	ENTLOA
1	3.3673	0.0000	3.3322	2.3026	1.6094	3.7842	2.8332	0.5361	0.3985	1.3176	3.2317
2	4.1897	1.7918	4.0943	3.0445	2.3026	4.5747	3.3322	0.1997	0.0578	1.5089	3.3900
3	4.9200	2.3979	4.8363	3.1355	2.8332	5.1761	4.0775	-0.0396	-0.1346	1.6367	3.9430
4	3.8501	1.6094	3.7377	2.3979	1.0986	4.1109	2.8332	-0.1339	-0.4553	1.4079	2.3780
5	4.2047	1.7918	4.1109	2.3979	1.0986	4.3944	3.2958	0.2464	0.1082	1.4948	3.4040
6	5.1240	3.0910	4.9836	3.4965	2.3026	5.3519	4.0431	-0.3919	-0.5507	1.6254	3.4923
7	4.6444	3.2958	4.3438	2.9957	2.0794	4.8828	3.6636	0.3483	0.2519	1.6254	3.9154
8	5.2933	3.6109	5.0876	3.8501	2.9444	5.5797	4.3438	0.0519	-0.0233	1.7703	4.3205
9	6.2005	4.0604	6.0753	4.8442	4.0775	6.5206	5.1985	-0.6362	-0.7108	1.9294	4.4877

CONTROL STRUCTURE ENTROPY
FOR C PROGRAMS

VARIABLE	N	MEAN	STD DEV	SUM	MINIMUM	MAXIMUM
LOCODE	9	4.609967	0.6979130	41.48970	3.637600	5.656000
DECL	9	2.521222	0.5175192	22.69100	1.791800	3.135500
NONDECL	9	4.195956	0.7048815	37.76360	3.258100	5.283200
BRACE	9	2.992744	0.8928902	26.93470	1.791800	4.219500
COMMENTS	9	3.048644	0.7281197	27.44680	2.302600	4.127100
BLANK	9	2.515300	0.7780832	22.63770	1.386300	3.610900
TOTAL	9	4.942789	0.6511411	44.48510	3.970300	5.826000
CONSTENT	9	2.423700	0.6310665	21.81330	1.609400	3.434000

PEARSON CORRELATION COEFFICIENTS / PROB > |R| UNDER H0:RHO=0 / N = 9

	LOCODE	DECL	NONDECL	BRACE	COMMENTS	BLANK	TOTAL
CONSTENT	0.92835	0.59440	0.95102	0.91013	0.00774	0.89001	0.83952
	0.0003	0.0914	0.0001	0.0007	0.9842	0.0013	0.0046

PEARSON CORRELATION COEFFICIENTS / PROB > |R| UNDER HO:RHO=0 / N = 9

	LOCODE	DECL	NONDECL	BRACE	COMMENTS	BLANK	TOTAL
LOCODE	1.00000	0.81961	0.99104	0.98490	0.29913	0.94904	0.97267
	0.0000	0.0068	0.0001	0.0001	0.4342	0.0001	0.0001
DECL	0.81961	1.00000	0.73955	0.86283	0.77162	0.86248	0.92752
	0.0068	0.0000	0.0228	0.0027	0.0149	0.0028	0.0003
NONDECL	0.99104	0.73955	1.00000	0.95816	0.18083	0.91257	0.93414
	0.0001	0.0228	0.0000	0.0001	0.6415	0.0006	0.0002
BRACE	0.98490	0.86283	0.95816	1.00000	0.38467	0.98308	0.98335
	0.0001	0.0027	0.0001	0.0000	0.3067	0.0001	0.0001
COMMENTS	0.29913	0.77162	0.18083	0.38467	1.00000	0.43154	0.50961
	0.4342	0.0149	0.6415	0.3067	0.0000	0.2461	0.1611
BLANK	0.94904	0.86248	0.91257	0.98308	0.43154	1.00000	0.96406
	0.0001	0.0028	0.0006	0.0001	0.2461	0.0000	0.0001
TOTAL	0.97267	0.92752	0.93414	0.98335	0.50961	0.96406	1.00000
	0.0001	0.0003	0.0002	0.0001	0.1611	0.0001	0.0000

OBS	LOCODE	DECL	NONDECL	BRACE	COMMENTS	BLANK	TOTAL	CONSTENT
1	4.1431	2.7081	3.5835	2.4849	3.7377	2.0794	4.7274	1.6094
2	4.4188	2.7726	3.8918	2.8904	3.9120	2.3979	4.9698	2.0794
3	4.9273	2.9957	4.4308	3.5264	4.1271	3.1781	5.4116	2.5649
4	3.6376	1.7918	3.2581	1.7918	2.3026	1.6094	3.9703	1.9459
5	4.4886	2.1972	4.0943	2.9957	2.3979	2.6391	4.7362	2.5649
6	5.6058	2.9957	5.2149	4.2195	3.0910	3.6109	5.8021	3.4340
7	3.9120	1.7918	3.6376	1.7918	2.3026	1.3863	4.1589	1.7918
8	4.7005	2.3026	4.3694	3.0445	2.4849	2.3026	4.8828	2.5649
9	5.6560	3.1355	5.2832	4.1897	3.0910	3.4340	5.8260	3.2581

ENTROPY LOADING METRIC
FOR C PROGRAMS

VARIABLE	N	MEAN	STD DEV	SUM	MINIMUM	MAXIMUM
LOCODE	9	4.609967	0.6979130	41.48970	3.637600	5.656000
DECL	9	2.521222	0.5175192	22.69100	1.791800	3.135500
NONDECL	9	4.195956	0.7048815	37.76360	3.258100	5.283200
COMMENTS	9	2.992744	0.8928902	26.93470	1.791800	4.219500
BRACE	9	3.049644	0.7281197	27.44680	2.302600	4.127100
BLANK	9	2.515300	0.7780832	22.63770	1.386300	3.610900
TOTAL	9	4.942789	0.6511411	44.48510	3.970300	5.826000
OBJTOTAL	9	3.850767	0.6317251	34.65690	2.944400	4.875200
AVOBEN	9	-0.026167	0.3242692	-0.23550	-0.512400	0.541600
RATIO	9	-0.164756	0.3566293	-1.48280	-0.686000	0.402200
SYSENT	9	1.665556	0.1296578	14.99000	1.446400	1.856800
ENTLOA	9	3.686033	0.6618569	33.17430	3.012200	4.765700

PEARSON CORRELATION COEFFICIENTS / PROB > |R| UNDER H0:RHO=0 / N = 9

	LOCODE	DECL	NONDECL	COMMENTS	BRACE	BLANK	TOTAL
OBJTOTAL	0.99473	0.81417	0.98974	0.96698	0.28961	0.91680	0.96372
	0.0001	0.0076	0.0001	0.0001	0.4497	0.0005	0.0001
AVOBEN	-0.32920	-0.69015	-0.24418	-0.36263	-0.88914	-0.35246	-0.49853
	0.3870	0.0396	0.5266	0.3375	0.0013	0.3522	0.1719
RATIO	-0.17775	-0.59058	-0.08873	-0.21687	-0.88560	-0.21412	-0.36148
	0.6473	0.0941	0.8204	0.5751	0.0015	0.5801	0.3391
SYSENT	0.97854	0.85488	0.96088	0.96557	0.39422	0.91424	0.97318
	0.0001	0.0033	0.0001	0.0001	0.2938	0.0006	0.0001
ENTLOA	0.85369	0.45892	0.89689	0.80613	-0.20074	0.75972	0.72510
	0.0034	0.2140	0.0010	0.0087	0.6045	0.0175	0.0271

PEARSON CORRELATION COEFFICIENTS / PROB > |R| UNDER H0:RHO=0 / N = 9

	LOCODE	DECL	NONDECL	COMMENTS	BRACE	BLANK	TOTAL
LOCODE	1.00000	0.81961	0.99104	0.98490	0.29913	0.94904	0.97267
	0.0000	0.0068	0.0001	0.0001	0.4342	0.0001	0.0001
DECL	0.81961	1.00000	0.73955	0.86283	0.77162	0.86248	0.92752
	0.0068	0.0000	0.0228	0.0027	0.0149	0.0028	0.0003
NONDECL	0.99104	0.73955	1.00000	0.95816	0.18083	0.91257	0.93414
	0.0001	0.0228	0.0000	0.0001	0.6415	0.0006	0.0002
COMMENTS	0.98490	0.86283	0.95816	1.00000	0.38467	0.98308	0.98335
	0.0001	0.0027	0.0001	0.0000	0.3067	0.0001	0.0001
BRACE	0.29913	0.77162	0.18083	0.38467	1.00000	0.43154	0.50961
	0.4342	0.0149	0.6415	0.3067	0.0000	0.2461	0.1611
BLANK	0.94904	0.86248	0.91257	0.98308	0.43154	1.00000	0.96406
	0.0001	0.0028	0.0006	0.0001	0.2461	0.0000	0.0001
TOTAL	0.97267	0.92752	0.93414	0.98335	0.50961	0.96406	1.00000
	0.0001	0.0003	0.0002	0.0001	0.1611	0.0001	0.0000

PEARSON CORRELATION COEFFICIENTS / PROB > |R| UNDER H0:RHO=0 / N = 9

	OBJTOTAL	AVOBEN	RATIO	SYSENT	ENTLOA
OBJTOTAL	1.00000	-0.34677	-0.19580	0.98329	0.84900
	0.0000	0.3606	0.6137	0.0001	0.0038
AVOBEN	-0.34677	1.00000	0.98737	-0.47910	0.20103
	0.3606	0.0000	0.0001	0.1919	0.6040
RATIO	-0.19580	0.98737	1.00000	-0.33518	0.35194
	0.6137	0.0001	0.0000	0.3779	0.3530
SYSENT	0.98329	-0.47910	-0.33518	1.00000	0.75794
	0.0001	0.1919	0.3779	0.0000	0.0180
ENTLOA	0.84900	0.20103	0.35194	0.75794	1.00000
	0.0038	0.6040	0.3530	0.0180	0.0000

OBS	LOCODE	DECL	NONDECL	COMMENTS	BRACE	BLANK	TOTAL	OBJTOTAL	AVOBEN	RATIO	SYSENT	ENTLOA
1	4.1431	2.7081	3.5835	2.4849	3.7377	2.0794	4.7274	3.4657	-0.2362	-0.4536	1.5969	3.0122
2	4.4188	2.7726	3.8918	2.8904	3.9120	2.3979	4.9698	3.6636	-0.3719	-0.5884	1.6552	3.0751
3	4.9273	2.9957	4.4308	3.5264	4.1271	3.1781	5.4116	4.0943	-0.5124	-0.6860	1.7451	3.4084
4	3.6376	1.7918	3.2581	1.7918	2.3026	1.6094	3.9703	2.9444	0.5416	0.4022	1.4464	3.3466
5	4.4886	2.1972	4.0943	2.9957	2.3979	2.6391	4.7362	3.6376	0.3007	0.1963	1.6273	3.8339
6	5.6058	2.9957	5.2149	4.2195	3.0910	3.6109	5.8021	4.6913	0.0381	-0.0173	1.8082	4.6741
7	3.9120	1.7918	3.6376	1.7918	2.3026	1.3863	4.1589	3.2958	0.0458	-0.1353	1.5435	3.1605
8	4.7005	2.3026	4.3694	3.0445	2.4849	2.3026	4.8828	3.9890	0.0152	-0.0912	1.7106	3.8978
9	5.6560	3.1355	5.2832	4.1897	3.0910	3.4340	5.8260	4.8752	-0.0564	-0.1095	1.8568	4.7657

VITA

William R. Torres

Candidate for the Degree of

Master of Science

Thesis: THE EFFECT OF SOFTWARE REUSABILITY ON INFORMATION THEORY BASED SOFTWARE METRICS

Major Field: Computing and Information Science

Biographical:

Personal Data: Born in San Juan, Puerto Rico, December 27, 1962, the son of William R. and Nelida Torres. Married to Maria A. Torres and father of Maria C. and Diana B. Torres.

Education: Graduated from Our Lady of Pilar High School, Rio Piedras, Puerto Rico, in May 1980; received Bachelor of Science Degree in Electrical Engineering from University of Puerto Rico at Mayaguez in May 1985; completed requirements for the Master of Science degree at Oklahoma State University in July 1990.

Professional Experience: Teaching Assistant, Department of Electrical Engineering, University of Puerto Rico at Mayaguez, August 1984, to December 1984. Physical Security Systems Engineer, Tinker Air Force Base (AFB) Oklahoma, June 1985, to July 1987. Local Area Network Design Engineer, Tinker AFB Oklahoma, July 1987, to August 1988. Branch Chief, Communications Support Branch, Wright Patterson AFB Ohio, January 1990, to present.

BIBLIOGRAPHY

[ADA83] Reference Manual for the Ada Programming Language, United States Department of Defense, ANSI/MIL-STD-1815A, January 1983.

[ALEXA64] Alexander, Christopher, Notes on the Synthesis of Form, Harvard University Press, Cambridge, Mass., 1964.

[BIGGE87] Biggerstaff, Ted and Richter, Charles, "Reusability: Framework, Assessment, and Directions," IEEE Software, March 1987, pp. 41-49.

[BOOCH86] Booch, Grady, Software Engineering with Ada, The Benjamin/Cummings Publishing Company Inc., Second Edition, 1986.

[CHANO73] Chanon, Robert N., "On a Measure of Program Structure," Ph.D. Dissertation, Department of Computer Sciences, Carnegie-Mellon University, Pittsburgh, PA, November 1973.

[CHEAT83] Cheatham, Jr., T. E., "Reusability Through Program Transformations," Proceedings ITT Workshop on Reusability in Programming, September 7-9, 1983, pp. 122-128.

[CHEN78] Chen, Edward T., "Program Complexity and Programmer Productivity," IEEE Transactions on Software Engineering, Vol. SE-4, No. 3, May 1978, pp. 187-194.

[CHENG84] Cheng, Thomas T., Lock, Evan D., and Prywes, Noah S., "Use of Very High Level Languages and Program Generation by Management Professionals," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 552-563.

[CONTE86] Conte, S. D., Dunsmore, H. E., and Shen, V. Y., Software Engineering Metrics and Models, The Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1986.

[GOGUE86] Goguen, Joseph A., "Reusing and Interconnecting Software Components," IEEE Computer, Vol. 19, February 1986, pp. 16-28.

[HALST79] Halstead, M. H., "Advances in Software Science," Advances in Computers, (Yovits, ed.), Vol. 18, Academic Press, New York, 1979, pp. 119-172.

[HARTM66] Hartmanis, J. and Stearns, R. E., Algebraic Structure Theory of Sequential Machines, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1966.

[KAISE87] Kaiser, Gail E. and Garlan, David, "Systems from Reusable Building Blocks," IEEE Software, July 1987, pp. 17-24.

[KERNI78] Kernighan, Brian W. and Ritchie, Dennis M., The C Programming Language, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[KERNI84] Kernighan, Brian W., "The UNIX System and Software Reusability," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 513-518.

[LETOV86] Letovsky, Stanley and Soloway, Elliot, "Delocalized Plans and Program Comprehension," IEEE Computer, May 1986, pp. 41-49.

[LUBAR86a] Lubars, Mitchell D., "Code Reusability in the Large Versus Code Reusability in the Small," ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 1, January 1986, pp. 21-27.

[LUBAR86b] Lubars, Mitchell D., "Affording Higher Reliability Through Software Reusability," ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 5, October 1986, pp. 39-42.

[MATSU84] Matsumoto, Yoshihiro, "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 502-513.

[MCCAB76] McCabe, J., "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 308-320.

[MILLE87] Miller, Webb, A Software Tools Sampler, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

[MOHAN81] Mohanty, Siba N., "Entropy Metrics for Software Design Evaluation," The Journal of Systems and Software 2, 1981, pp. 39-46.

[MOHNK86] Mohnkern, Gerald L. and Mohnkern, Beverly, Applied Ada, Tab Professional and Reference Books, 1986.

[PRIET87] Prieto-Díaz, Rubén and Freeman, Peter, "Classifying Software for Reusability," IEEE Software, January 1987, pp. 6-16.

[SAS85a] SAS User's Guide: Basics, Version 5 Edition, SAS Institute Inc., Box 8000, Cary, NC 27511, 1985.

[SAS85b] SAS User's Guide: Statistics, Version 5 Edition, SAS Institute Inc., Box 8000, Cary, NC 27511, 1985.

[SCHIL87] Schildt, Herbert, Advanced Turbo C, Osborne McGraw-Hill, New York, NY, 1987.

[SCHUT77] Schütt, Dieter, "On a Hypergraph Oriented Measure for Applied Computer Science," Proceedings of COMPCON, Washington, D.C., Fall 1977, pp. 295-296.

[SHANN64] Shannon, Claude E. and Weaver, Warren, The Mathematical Theory of Communication, The University of Illinois Press, Urbana, Ill., 1964.

[SHOOM83] Shooman, Martin L., Software Engineering: Design, Reliability, and Management, McGraw-Hill Book Company, New York, 1983.

[SHUMA89] Shumate, Ken, Understanding Ada with Abstract Data Types, John Wiley and Sons, Inc., New York, NY, Second Edition, 1989.

[SOMME89] Sommerville, Ian, Software Engineering, Addison-Wesley Publishing Co., Third Edition, 1989.

[STEVE74] Stevens, W. P., Myers, G. J., and Constantine, L. L., "Structured Design," IBM Systems Journal, Vol. 2, 1974, pp. 115-139.

[TRACZ88] Tracz, Will, "Software Reuse Maxims," ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 5, October 1988, pp. 28-31.

[VANEM70] van Emden, M. H., "Hierarchical Decomposition of Complexity," Machine Intelligence 5, 1970, pp. 361-380.